KATHOLIEKE UNIVERSITEIT
LEUVEN

Faculty of Economics and Applied Economics

Using rule extraction to improve the comprehensibility of predictive models

Johan Huysmans, Bart Baesens and Jan Vanthienen

DEPARTMENT OF DECISION SCIENCES AND INFORMATION MANAGEMENT (KBI)

KBI 0612

Using Rule Extraction to Improve the Comprehensibility of Predictive Models

Johan Huysmans, Bart Baesens and Jan Vanthienen

Katholieke Universiteit Leuven
Department of Decision Sciences and Information Management
Naamsestraat 69, 3000 Leuven, Belgium

Abstract

Whereas newer machine learning techniques, like artificial neural networks and support vector machines, have shown superior performance in various benchmarking studies, the application of these techniques remains largely restricted to research environments. A more widespread adoption of these techniques is foiled by their lack of explanation capability which is required in some application areas, like medical diagnosis or credit scoring. To overcome this restriction, various algorithms have been proposed to extract a meaningful description of the underlying 'black box' models. These algorithms' dual goal is to mimic the behavior of the black box as closely as possible while at the same time they have to ensure that the extracted description is maximally comprehensible.

In this research report, we fist develop a formal definition of 'rule extraction' and comment on the inherent trade-off between accuracy and comprehensibility. Afterwards, we develop a taxonomy by which rule extraction algorithms can be classified and discuss some criteria by which these algorithms can be evaluated. Finally, an in-depth review of the most important algorithms is given. This report is concluded by pointing out some general shortcomings of existing techniques and opportunities for future research.

1 Introduction

1.1 Definition

Whereas there already exist various definitions of 'rule extraction', we believe that most of these are too restrictive. For example, Craven [13] defines rule extraction as:

"Given a trained neural network and the data on which it was trained, produce a description of the network's hypothesis that is comprehensible yet closely

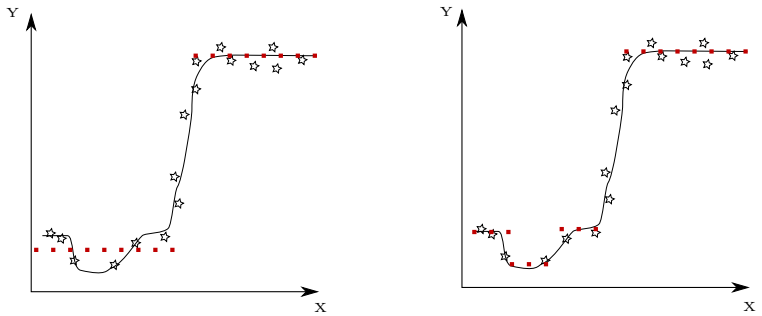
approximates the network's prediction behaviour."

which is reformulated in [10] as:

"In the scenario of high dimensional feature spaces, given a trained SVM and the data on which it was trained, produce a description of the SVM's hypothesis that is understandable yet closely approximates the SVM's prediction behavior"

Both definitions view the process of rule extraction from their specific field of interest, respectively neural networks and SVMs. However, the above definitions of rule extraction can be formulated more generally as:

> *"Given an opaque predictive model and the data on which it was trained, produce a description of the predictive model's hypothesis that is understandable yet closely approximates the predictive model's behavior"*



(a) Better Comprehensibility (b) Better Accuracy

Figure 1: Trade-off between Accuracy and Comprehensibility (full line=original model, dotted line=extracted rules)

A first issue to notice from this definition is that although the process is coined 'rule extraction', it is not required that the returned description consists of rules. We consider any method that delivers a human-comprehensible description of the underlying model as a rule extraction technique. Although most algorithms will indeed provide a description consisting of rules, there exist techniques that provide descriptions in different formats such as decision trees, finite state machines or graphical models. The term 'white box extraction' might therefore be more appropriate than the widespread term 'rule extraction'.

From the above definitions, one can also observe the inherent duality between comprehensibility and accuracy[1]. The extracted description should be comprehensible but at the same time approximate the underlying model as closely as

[1] As explained below, 'fidelity' might be a more appropriate term

possible. Which of both goals is favored usually depends on the specific requirements of the application. For example, in Figure 1, a regression model (full line) is built from the available sample observations. In the left figure, a simple model (dotted line) of the following form is extracted:

if X is smaller than threshold then Y is Value_1
else Y is Value_2

While this rule set provides a relatively good approximation of the underlying model, there are some minor deviations between the original model and the extracted rules. Figure 1(b) shows a rule set that better approximates the underlying model.

if X is smaller than threshold_1 then Y is Value_1
if X $\in$ [threshold_1,threshold_2] then Y is Value_2
if X $\in$ [threshold_2,threshold_3] then Y is Value_3
else Y is Value_4

This rule set is however more elaborate and contains twice the amount of rules as the first rule set. Which of both rule sets is preferred, usually depends on the specific requirements of the application at-hand.

During our review of rule extraction algorithms, this trade-off between accuracy and comprehensibility will be encountered several times. Most algorithms deal with the duality by allowing users to set some parameters to specify a desired level of accuracy or a maximum level of complexity.

1.2 Motivation

At this point, one might start wondering why rule extraction is important. If it is of major concern to have a good understanding of the predictive model, it seems logical to avoid the intermediate step of creating an opaque model and to start directly with a 'white box' model.

The main motivation is that a well trained intermediate model can often better represent the data than the data itself by filtering out the noise that is present in the samples. Additionally, the use of the intermediate 'black box' model allows the creation of 'artificial' data examples for those regions of the input space that are covered by only a small number of sample points. Finally, several benchmarking studies [5, 54, 57] have shown that in many application areas opaque models achieve better performance than 'white box' models. In situations where performance is a crucial issue, the opaque models are therefore the preferred choice for implementation in the decision process but rule extraction can be adopted to verify the knowledge encoded in these models.

This knowledge verification is crucial in many application areas and sometimes even legally required. For example, the Equal Credit Opportunity Act [1] is a US federal law which prohibits creditors from certain forms of discrimination. Under this law, financial institutions are required to provide specific

reasons in case an application is rejected: indefinite and vague reasons for denial are illegal [33]. An unacceptable motivation is for example: "You didn't receive enough points on our credit scoring system", which is pretty much the only possible explanation that can be provided by a 'black box' scoring system when no rule extraction is performed.

Similar requirements can also be found in the medical domain, where users are reluctant to use 'black box' computer-aided diagnosis (CAD) systems that can influence patient treatment [21]. The ability to generate even limited explanations is essential for user acceptance of such systems.

Apart from the explanation capability, several other reasons that underline the importance of rule extraction are mentioned in [3]. Automatic knowledge acquisition is one of them. Constructing and debugging a knowledge base for a decision support system is a difficult and time-consuming task. Automatic rule-extraction can considerably facilitate the burden of maintaining such knowledge base. Other possible motivations for rule extraction are the induction of scientific theories or the study of the generalization behavior of the underlying model.

1.3 Rule Types

As discussed above, most of the extraction algorithms generate rules to describe the underlying model's hypothesis. Many different types of rules exist. In this section, we provide an overview of several frequently used rule types.

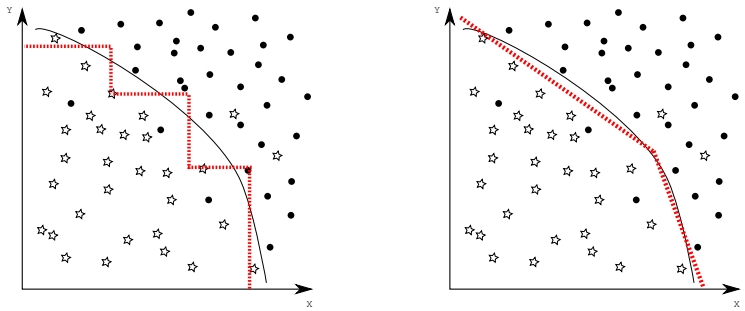
The most widespread rules are without any doubt **propositional if-then rules**. The condition part of a propositional rule is a boolean combination of conditions on the input variables. Whereas the condition part can contain conjunctions, disjunctions and negations, most algorithms will return rules that only contain conjunctions. An example of such a rule is: 'if X=a and Y=b then Class=1', with X,Y input variables and a,b possible values of these variables. For continuous input variables, the conditions are usually specified as restrictions on the allowed values, e.g 'X $\in [c_1, c_2]$' or 'X $> c_3$' with c_1, c_2, $c_3 \in \mathbb{R}$.

Most algorithms will ensure that the condition parts of each rule demarcate separate areas in the input space: i.e. the rules are mutually exclusive. Therefore, only one rule can be satisfied when a new observation is presented and the rule that has fired will be the only one used for making the classification (or regression) decision. However, some algorithms will allow multiple rules to fire for the same instance. This requires an additional mechanism to combine the individual predictions. For example, [10] associates a confidence factor with each rule and rules that fire with a large confidence factor have a greater impact on the final decision. Sorting the rules and allowing only the first firing rule to decide is another mechanism, applied in [51].

M-of-N rules are closely related to propositional rules. They are expressions of the form 'if {at least/exactly/at most} M of the N conditions $\{C_1, C_2, \ldots, C_N\}$ are satisfied then Class=1'. It is usually straightforward to convert M-of-N rules into propositional rules. For example, the M-of-N rule 'if exactly 2-of-{X=a,Y=b,Z=c} then Class=1' is logically equivalent to 'if ((X=a and Y=b) or (X=a and Z=c) or (Y=b and Z=c)) then Class=1'. Observe that

for M=1, the rules can be written as a number of disjunctions while for M=N the rules can be expressed as a conjunction of the conditions. The use of M-of-N rules was first proposed in [56]. Other algorithms that provide M-of-N rules as result of their extraction process are the popular TREPAN [15] and FERNN [44].

A third type of rules, which can not be transformed straightforwardly into propositional rules, are **oblique rules**. Oblique rules represent piecewise discriminant functions and are usually represented as follows: 'if (c_1X+c_2Y > c_3) and (c_4X+c_5Z > c_6) and ... then Class=1' with $c_1, \ldots, c_6 \in \mathbb{R}$. In comparison with propositional rules, oblique rules are usually more difficult to understand. However, oblique rules have the advantage that they can create decision boundaries that are non-parallel with the axes of the original input space. In [48], it is argued that oblique rules may therefore require fewer conditions than propositional rules. An example is shown in Figure 2.



(a) Parallel Boundaries (b) Oblique Boundaries

Figure 2: Parallel versus Oblique Boundaries

In the left part of this figure, the decision boundary is approximated with three propositional rules (or as one rule with three disjunctions in the condition part). The right part shows the approximation of the decision boundary by an oblique rule. We can observe that the same level of accuracy would require a large number of propositional rules.

However, when categorical attributes are present in the data oblique rules become even less interpretable. In [43], this problem is alleviated by the extraction of a hierarchical rule set that combines opaque and propositional rule characteristics. Only rules that are low in the hierarchy have conditions that involve linear combinations of continuous attributes while the conditions of all the other rules contain solely discrete attributes. It is argued that such rule conditions greatly increase the comprehensibility of the rules.

Very similar to oblique rules, but slightly more complex, are **equation rules**. This type of rules was proposed in [36], and contain a polynomial equation in the condition part. An example is : 'if $c_1X^2 + c_2Y^2 + c_3XY + c_4$X+$c_5$Y < c_6 then Class=1' with $c_1, \ldots, c_6 \in \mathbb{R}$. Equation rules were however only used in

that particular study. It is difficult to understand them and they therefore contribute little to the interpretation of the underlying model.

A fifth type of rules are **fuzzy rules**. An example of a fuzzy classification rule is: 'if X is low and Y is medium then Class=1', whereby low and medium are fuzzy sets with corresponding membership functions. Fuzzy rules are believed to be both comprehensible and user-friendly because the rules are expressed in terms of linguistic concepts, which are easy to interpret for the human expert.

A final category are rules expressed in **First Order Logic**, i.e. rules that can contain quantifiers and variables. At this moment, we are however not aware of any algorithms that can extract such rules directly and we will therefore not cover this category in more detail.

2 Taxonomy of Rule Extraction Algorithms

In this section, we develop a taxonomy that can be used for discussion and evaluation of rule extraction algorithms. Although there already exists a widely used taxonomy, the ADT -Taxonomy[2] proposed in [3], we believe that a new classification schema may be required to cover the algorithms that extract rules from black box models other than neural networks. The ADT-Taxonomy includes several elements, e.g. translucency and portability, that are specifically defined for neural networks. This makes it less suitable for classifying extraction algorithms that assume other models, such as SVMs (e.g. [36]).

The taxonomy that we propose, contains three main criteria for evaluation of algorithms: the **scope of use**, the **type of dependency on the black-box** and the **format of the extracted description**. The latter two of these criteria are also present in the ADT-taxonomy, but as explained below with a different interpretation.

The first dimension concerns the scope of use of an algorithm: either **regression or classification**. While there are a few algorithms that are applicable for both types of problems, e.g. G-REX [25] and ITER [23], the majority is specifically designed for either one of those. This dimension was included in the taxonomy because it gives an immediate indication about the appropriateness of an algorithm for a specific application. The inclusion of this criterion in the ADT-Taxonomy was also proposed in [4].

The second dimension focuses on the dependency of the extraction algorithm on the underlying black box: **independent versus dependent** algorithms. We define a rule extraction algorithm as **independent** if it is totally independent of the underlying black box model. It is therefore possible to use the same algorithm in combination with different types of opaque models, such as neural networks, support vector machines or an ensemble learner. The only requirement is that the underlying model can be queried. It acts as an oracle that provides predictions for the observations that it receives. The basic idea is to use the 'black box' model as an example-generator from which the algorithm can learn. This added layer of complexity -first training a model on the data and

[2]ADT= after the authors Andrews, Diederich and Tickle

then extracting rules from the model- has often proven advantageous in comparison with direct rule extraction from the data points. The motivation is that a well trained model can often better represent the data than the original data set [33]. Additionally, the use of the 'black box' model allows the creation of 'artificial' data examples for those regions of the input space where not enough original data points are available.

Algorithms that use information about the inner-workings of the underlying model are **dependent**. A dependent algorithm can only be applied if the underlying model is of a certain form. Knowledge about the inner-workings of the underlying model can then be used for creation of rules. For example, in [45] the behavior of a regression neural network is converted into regression rules by approximating the activation function of the hidden neurons by a combination of linear functions. For support vector machines, dependent techniques usually rely on the support vectors for the extraction of their rules (e.g., [36]). The main disadvantage of these dependent algorithms is that these techniques pose strict restrictions on the underlying models: they assume that certain activation functions are used or that the architecture of the neural network follows a certain specification.

Readers familiar with the literature on rule extraction, will probably see the similarity of the above distinction with the translucency dimension in the ADT-Taxonomy [3]. The translucency dimension is used to describe the relationship between the extraction algorithm and the internal architecture of the underlying neural network. It makes a distinction between **decompositional** and **pedagogical** techniques[3]. A decompositional technique focuses on the individual units (neurons) within the network and aggregates the rules extracted at the individual levels into a composite rule set. Pedagogical techniques are different because they do not extract rules for the individual neurons but create rules that map the inputs directly into outputs. During our evaluation of the different rule extraction algorithms, we will observe that there is a large overlapping between dependent and decompositional algorithms and also between independent and pedagogical algorithms. The overlapping is however not complete. For example, the VIA-algorithm [52] is an example of a pedagogical algorithm. It creates rules by propagating intervals through the network. Because it uses information about the weights of the neural network during the propagation step, VIA can not be applied to other models than neural networks. VIA is therefore a dependent technique.

In the rest of this report, we will use all of the above terms to refer to the translucency of the algorithms. If we state that an algorithm is decompositional then it automatically implies that the algorithm is dependent. Unless explicitly stated otherwise, we will also assume that the term 'pedagogical' implies the 'independence' of the extraction technique.

While the above distinction between dependent and independent algorithms is useful for the classification of rule extraction techniques, a third criterion that focuses more on the obtained rules might be worthwhile: predictive versus de-

[3]We do not cover the eclectic and compositional approaches

scriptive algorithms. We call a rule extraction algorithm **predictive** when the extracted rules allow the analyst to make a prediction for each possible observation from the input space. More specifically, every possible input observation should be covered by exactly one rule, which means that the extracted rules should be both exclusive and exhaustive. By having both exclusive and exhaustive rules, the order in which the rules are processed is of no importance. For classification problems, exhaustivity is often realized by creating only rules for the minority class and adding an additional rule that specifies a default class when the input observation was not covered by any of the other rules. While this imposes a partial ordering on the rules, we consider these rule sets as predictive because it is relatively straightforward to convert them into predictive rules. Algorithms that extract decision trees are also examples of predictive techniques.

If the rule sets created by a rule extraction algorithm are not predictive, then we call it a **descriptive algorithm**. The resulting rules are either not-exhaustive or not-exclusive. Non-exhaustivity might provide problems as there will be input observations for which none of the rules fires and no forecast can be delivered. Non-exclusivity might also provide problems because observations can be covered by multiple rules when non-exclusive rules are present. This requires an additional mechanism to combine the individual predictions. For example, [10] associates a confidence factor with each rule and rules that fire with a large confidence factor have a greater impact on the final decision. Sorting the rules and allowing only the first firing rule to decide is another mechanism, applied in [51]. To avoid these problems, users will often prefer the use of predictive algorithms over descriptive algorithms.

An overview of this taxonomy together with the categorization of several algorithms is shown in Table 1.

	Independent Algorithms		**Dependent Algorithms**	
	Classification	**Regression**	**Classification**	**Regression**
Predictive Algorithms	CART, C4.5, TREPAN [15], G-REX [25],BIO-RE[51]	ITER, G-REX [25], CART, ANN-DT [42]	Barakat et al. [7], Fung et al. [21], FERNN [44], NeuroLinear [48], RE-RX [43]	REFANN[45], RN2[41]
Descriptive Algorithms	STARE[60], REFNE [61], GEX [31], BUR [10]		SVM+Prototype [36], VIA [52]	

Table 1: Taxonomy of Rule Extraction Algorithms

3 Overview of rule Extraction Algorithms

3.1 Evaluation Criteria

In existing surveys [3, 16, 35], rule extraction algorithms are evaluated by a number of criteria. Besides the characteristics used during creation of the taxonomy, such as the expressive power of the extracted rules and applicability of the algorithm, several other criteria appear in almost all of these surveys:

- Quality of the extracted rules
- Scalability of the algorithm
- Consistency of the algorithm

We will discuss each of these criteria in greater detail below.

3.1.1 Quality of the Extracted Rules

Quality of the extracted rules is considered to be the most important evaluation criterion for rule extraction algorithms [35]. Many different aspects contribute to the aspect of rule quality: accuracy, fidelity, comprehensibility,...

The concept of **accuracy** describes whether the extracted rules can make correct predictions for previously unseen test examples. **Fidelity** is closely related to accuracy, and measures the ability of the extracted rules to mimic the behavior of the model from which they were extracted.

Let $\{\mathbf{x}_i, y_i\}_{i=1}^N$ represent observations with their corresponding label and y_i^{BB} and y_i^{WB} the predictions made by respectively the underlying (Black Box) model and the extracted rules (White Box).

For classification problems, accuracy is usually measured as the percentage of correctly classified observations. More formally, the percentage of observations for which y_i and y_i^{WB} are the same. Fidelity is then expressed as the percentage of observations for which y_i^{BB} and y_i^{WB} are the same.

$$accuracy^{WB} = Prob(y_i = y_i^{WB} | \mathbf{x}_i \in X) \tag{1}$$

$$fidelity^{WB} = Prob(y_i^{BB} = y_i^{WB} | \mathbf{x}_i \in X) \tag{2}$$

For regression problems, accuracy is usually measured by the mean-absolute or mean-squared error. These can also be adopted to measure the fidelity between the underlying model and the rules extracted from it. More formally, using the definition of mean-absolute error (MAE):

$$accuracy^{WB} = \frac{1}{N} \sum_{i=1}^{N} |y_i - y_i^{WB}| \tag{3}$$

$$fidelity^{WB} = \frac{1}{N} \sum_{i=1}^{N} |y_i^{BB} - y_i^{WB}| \tag{4}$$

In [59], it is argued that in certain situations it might be impossible to extract rules that have both high accuracy and high fidelity: a situation that is called the fidelity-accuracy dilemma. Consequently, authors must make a choice for either fidelity or accuracy. We are convinced that this dilemma is mainly caused by a badly trained black box model. When extracting rules from a largely overfitted model one might indeed face this problem, but if the underlying model correctly identifies the decision boundaries we see no reason for this dilemma to arise.

A third factor that is of major importance in determining the quality of the extracted rules is their **comprehensibility**. Although the main motivation of rule extraction is to obtain a comprehensible description of the underlying model's hypothesis, this aspect of rule quality is often overlooked. We believe that this is mainly due to the subjective nature of comprehensibility, which can not be measured independently of the person using the system [35]. Prior experience and domain knowledge of this person play an important role in comprehensibility. This contrasts with accuracy or fidelity that can be considered as properties of the rules and which can be evaluated independently of the users.

In most studies, model complexity is used as a proxy for model comprehensibility. The number of rules, the average number of antecedents, the number of leaf nodes of the decision tree, etc. are hereby used as indicators for the model comprehensibility.

3.1.2 Scalability of the algorithm

One desirable characteristic of a rule extraction algorithm is that it can be put in practice on a wide range of applications. This means that an algorithm should not only be able to deal with toy problems, but that it should also remain applicable to large-scale problems, i.e. when faced with a large number of examples or input features. Based on [16], scalability in the context of rule extraction can be defined as:

Scalability refers to how the running time of a rule-extraction algorithm and the comprehensibility of its extracted models vary as a function of such factors as the underlying model, the size of the training set and the number of input features

Besides including the traditional notion of scalability, i.e. running time or **algorithmic complexity**, the above definition also places emphasis on the aspect of comprehensibility. The extracted model should remain comprehensible, even if there are many input features and/or training examples. Let us denote this aspect of scalability as **non-algorithmic or comprehensibility complexity**. During the in-depth review of several prominent algorithms, we will observe that many of these algorithms were not developed with non-algorithmic complexity in mind. For example, most neural network decompositional algorithms construct rules based on the weights within the neural network and the size of the extracted rule set is usually proportional to the total number of weights [16]. Feature selection and severe pruning of weights and hidden neu-

rons is therefore crucial to obtain a comprehensible description. Pruning is also necessary to reduce the running time of these algorithms as most of them show worst-case exponential behavior [2].

3.1.3 Consistency of the algorithm

A final issue in the evaluation process is the **consistency** of the algorithm. There are however multiple definitions of this concept. In [3, 56], an algorithm is deemed consistent if under different training sessions, rule sets are generated that produce the same classification of unseen examples. Slightly different is the definition in [35] which labels consistency as the ability of an algorithm to extract rules with the same degree of accuracy under different training sessions. Whereas an algorithm that is consistent according to the first definition is automatically also consistent according to the second definition, the opposite is not necessarily true: rule sets with similar accuracy face the same number of misclassifications but the actual misclassifications can be different. Both definitions are similar in the fact that they only focus on the predictions from the extracted rules and ignore the rules themselves. According to the definition of consistency by Johansson et al. [27], similarity of the extracted rules is however the key aspect:

An algorithm is consistent if it extracts similar rules every time it is applied to the same data set.

However, the author immediately points to the difficulty associated with this definition: there is no straightforward definition of similarity that is applicable for the wide range of representations used in rule extraction. In the discussion below we present a method that we believe is general enough to measure the similarity between two rule sets (or other representation forms) for a given number of observations.

We believe that it is desirable for a measure of similarity σ between two rule sets to have the following properties:

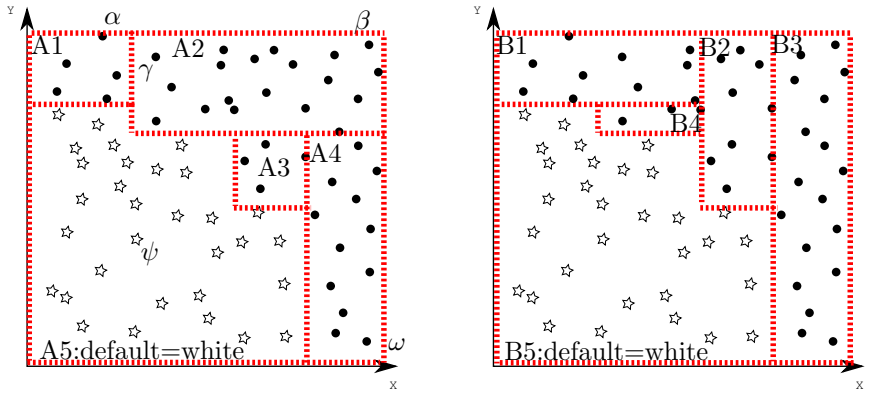
- similarity between A and B should be equal to similarity between B and A (symmetric)
  $\sigma(A, B) = \sigma(B, A)$
- If the two rule sets are exactly the same then the similarity should be maximal.
  $\sigma(A, A) = 1$
- If the two rule sets provide different classifications for each input observation then the similarity should be minimal
  $\sigma(A, \neg A) = 0$

We propose the following algorithm to calculate the similarity σ between two rule sets A and B, given N input observations.

1. Assign a unique identifier to each rule of A and B
2. For each observation: find the prediction made by set A and B and the identifiers of the rules of A and B that were used to make this prediction
3. For each rule r in A and B. Find the observations for which the classification decision was based on r. Denote this number of observations with N_r. For each rule in the other rule set that predicts the same class as r, find the identifier of the rule s that fired most frequently for these observations. Denote the number of times that rule s fired as $N_r^{opposite}$
4. Use as similarity measure the sum of all the $N_r^{opposite}$ divided by the sum of all N_r.

$$\sigma(A,B) = \frac{\sum_{r \in A,B} N_r^{opposite}}{\sum_{r \in A,B} N_r} = \frac{1}{2N} \sum_{r \in A,B} N_r^{opposite} \tag{5}$$

We will clarify the above algorithm with the example shown in Figure 3. An algorithm has provided two rule sets to divide the BLACK from the WHITE observations. We first assign each rule a unique identifier: A1,...,A5 and B1,...,B5. For this example, we will assume that all rules, except A5 and B5, predict the class to be BLACK. We can observe from the figure that A and B create the same decision boundary between the two classes and they will therefore make exactly the same predictions for all observations. This means that they are completely consistent according to the first two definitions. We will show how the rule sets are only partially consistent when applying the third definition and the algorithm described above.



(a) Rule Set A (b) Rule Set B

Figure 3: Consistency

In the second step of the algorithm Table 2 is constructed. For each observation, this table contains the predictions made by each rule set and the rule identifier of the rule that fired for the observation.

This table is used during the third step of the algorithm. For each rule, we first query the table to find out how many times the rule fired, e.g. $N_{A1} = 5$ and $N_{B1} = 10$. Afterwards, the algorithm looks for the rule in the other rule set that provides the same classification for most of these observations. In this example, all observations are classified the same by both rule sets, thus we only have to find the rule in the other rule set that fires the most. Because all 5 examples classified by rule A1 are classified by rule B1, $N_{A1}^{opposite} = 5$. For the 10 observations classified by rule B1, we can observe that half are classified by A1 and the other half by A2: $N_{B1}^{opposite}$=5. The same calculation is repeated for all the rules and the results of this step are shown in Table 3. The division of the totals in this table gives us a similarity estimate of 80%. It is not easy to give an intuitive meaning to this number, but the above procedure can be considered more or less as assigning to each rule a corresponding rule of the other rule set and the similarity measure is then the proportion of examples that were classified by corresponding rules.

	Rule Set A		**Rule Set B**	
Observation	Prediction	Rule	Prediction	Rule
α	BLACK	A1	BLACK	B1
β	BLACK	A2	BLACK	B3
γ	BLACK	A2	BLACK	B1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
ψ	WHITE	A5	WHITE	B5
ω	BLACK	A4	BLACK	B3

Table 2: Consistency: step 2

It is straightforward to alter the above algorithm to make it applicable to other representation forms, such as decision trees or decision tables. The only required change is in the first step of the algorithm. Instead of the assignment of an identifier to each rule, we will assign identifiers to each leaf-node for decision trees or to each column if the representation format is a decision table. The only assumption for the remainder of the algorithm is that each classification decision is based on exactly one identifier, i.e. the rules must be mutually exclusive (non-overlapping). This requirement is automatically fulfilled by various representation formats, such as decision trees and decision tables.

The proposed consistency measure is also not limited to classification problems only. With minor changes, the above algorithm can also be applied to measure consistency on regression problems. However, this requires an additional parameter s to indicate the desired level of sensitivity. Instead of requiring the corresponding rules to predict the same class, the sensitivity measure is used with regression problems to decide whether the predictions of corresponding

Rule	N_{Rule}	$N_{Rule}^{opposite}$
A1	5	5(B1)
A2	19	6(B3)
A3	4	4(B2)
A4	12	12 (B3)
A5	31	31 (B5)
B1	10	5(A1,A2)
B2	9	5(A2)
B3	18	12(A4)
B4	3	3 (A2)
B5	31	31 (A5)
SUM	142	114

Table 3: Consistency: step 3

rules are similar enough. Therefore, in the third step we would replace "For each rule in the other rule set that predicts the same class as r" by "*For each rule in the other rule set that provides the same forecast within s-difference as r*".

In most application areas, accuracy is probably much more important than consistency: the individual rules are unimportant as long as the rule set provides the correct classification. One might even argue that algorithms with low consistency are preferred as they might provide different views on the same data. In other domains the opposite situation may be encountered: consistency plays a pivotal role when explicatory power is the main requirement. Johansson [27] formulates this as follows: "it is very hard to give any significance to a specific rule set if the extracted rules vary significantly between runs."

3.2 Independent Algorithms

The main advantage of this category of algorithms is their independence of the underlying black box model. Although most algorithms in this category were originally conceived to extract rules from neural networks they remain applicable when the underlying model changes to a support vector machine or any other black box technique.

3.2.1 Rule Learners: CN2

Many algorithms are capable of learning rules directly from a set of training examples, e.g. CN2 [11], AQ [34], RIPPER [12]. Because of their ability to learn rules directly from data, these algorithms are not considered to be rule extraction techniques in the strict sense of the word. However, these algorithms can also be used to extract a human-comprehensible description from opaque models. When used for this purpose, the original target values of the training

```
SEQUENTIAL-COVERING(Class, Attributes, Examples, Threshold)
1   RuleSet= ∅
2   Rule =LEARN-ONE-RULE(Class, Attributes, Examples)
3   While (Performance(Rule)>Threshold)
4      RuleSet=RuleSet ∪ Rule
5      Examples = Examples - {examples correctly classified by Rule}
6      Rule= LEARN-ONE-RULE(Class, Attributes, Examples)
7   End While
8   Sort RuleSet according to performance of the rules
9   Return RuleSet
```

Figure 4: Sequential Covering

examples are modified by the predictions made by the black box model and the algorithm is then applied on this modified data set.

In this section, we discuss a general class of rule learners: sequential covering algorithms. This series of algorithms extracts a rule set by learning one rule, removing the data points covered by that rule and reiterating the algorithm on the remainder of the data. The general outline of sequential covering algorithms is given in Figure 4.

Starting from an empty rule set, the sequential covering algorithm first looks for a rule that is highly accurate for predicting a certain class. If the accuracy of this rule is above a user-specified threshold, then the rule is added to the set of already found rules and the algorithm is repeated over the rest of the examples that were not classified correctly by this rule. If the accuracy of the rule is below this threshold the algorithm will terminate. Because the rules in the rule set can be overlapping, the rules are first sorted according to their accuracy on the training examples before they are returned to the user. New examples are classified by the prediction of the first rule that was triggered.

It is clear that in the above algorithm, the subroutine LEARN-ONE-RULE is of crucial importance. The rules returned by the routine must have a good accuracy but do not necessarily have to cover a large part of the input space. The exact implementation of LEARN-ONE-RULE will be different for each algorithm but usually follows either a bottom-up or top-down search process. If the bottom-up approach is followed, the routine will start from a very specific rule and drops in each iteration the attribute that least influences the accuracy of the rule on the set of examples. Because each dropped condition makes the rule more general, the search process is also called specific-to-general search. The opposite approach is the top-down or general-to-specific search: the search starts from the most general hypothesis and adds in each iteration the attribute that most improves accuracy of the rule on the set of examples. This approach was followed in the CN2 algorithm [11] which is shown in Figure 5.

This LEARN-ONE-RULE implementation starts from the most general hypothesis and performs a beam search with a beam width of K. At each iteration

```
LEARN-ONE-RULE(Examples,K)
 1   BestHypothesis= ∅
 2   Candidates=BestHypothesis
 3   While (Candidates not empty) do
 4     AllConstraints={(A=V)| A is an attribute and V is a value of A}
 5     NewCandidates=∅
 6     For each Candidate C in Candidates
 7       For each Constraint R in AllConstraints
 8         SpecializedCandidate=Specialize C by adding R
 9         NewCandidates=NewCandidates ∪ SpecializedCandidate
10       End For
11     End For
12     Remove Duplicates and Inconsistencies from NewCandidates
13     For each C in NewCandidates
14       Calculate Performance of C on Examples
15       Update BestHypothesis if performance of C is better
16     End For
17     Candidates=K best members of NewCandidates
18   End While
19   Return Rule: "If BestHypothesis then Prediction"
```

Figure 5: Learn-One-Rule: CN2 implementation

the best K hypotheses are remembered and used as starting point for specialization during the next iteration. When the algorithm terminates, it will return the best hypothesis it has found together with the class label that corresponds to the majority class of the examples covered by this hypothesis.

As can be observed from line 4, the algorithm assumes that the attributes are categorical and creates tests of the form: Atribute=value. Consequently, for continuous attributes a discretization is required. Often this discretization will be performed once to the entire data set before the sequential covering algorithm is executed (global discretization). The same intervals will therefore appear in multiple rules which might be beneficial for the interpretability of the rule set. A second option is the application of local discretization. Local discretization is performed during the subroutine LEARN-ONE-RULE and is based on the examples that were provided as input to this routine. The same variable will therefore be discretized several times during execution of the algorithm and different interval conditions will appear in the rule set.

3.2.2 Decision Trees: C4.5 and CART

Decision trees [9, 37, 29] are widely used in predictive modeling. A decision tree is a recursive structure that contains a combination of internal and leaf nodes. Each internal node specifies a test to be carried out on a single variable and its branches indicate the possible outcomes of the test. An observation can

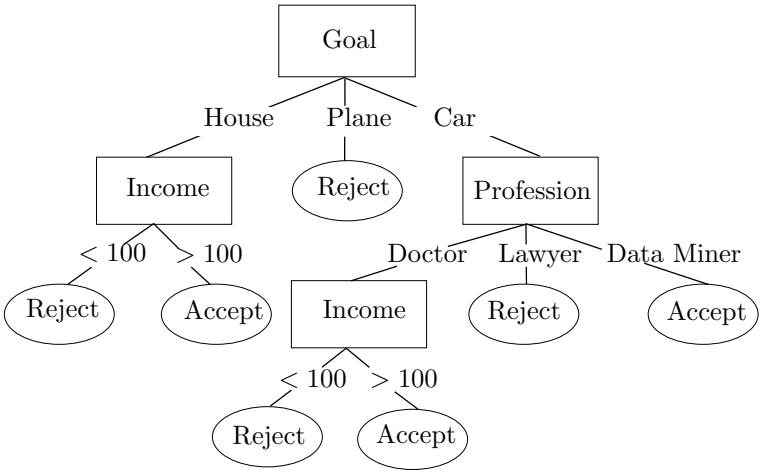


Figure 6: An Example Decision Tree

be classified by following the path from the root towards a leaf node. At each internal node, the corresponding test is performed and the outcome indicates the branch to follow. With each leaf node, a value or class label is associated. An example decision tree is shown in Figure 6. This fictious tree can be used by financial institutions to decide whether or not to accept loan applications.

Just like the rule learners discussed in the previous section, decision trees are usually constructed directly from the available training observations and are therefore not considered to be rule extraction techniques in the strict sense of the word. However, it is also possible to apply these techniques for pedagogical rule extraction by replacing the original targets with the target values provided by the trained black box model. The tree can then be trained on these new data points. If trees are used as an extraction technique, the observations that were wrongly predicted by the underlying model are usually not used during training. Additionally, random instances can be created and added to the training set. For example, in [6] a support vector machine is built from the original training data. A second data set is randomly generated with class labels for each instance provided by the support vector machine. Both data sets are then combinedly used to train a C5 decision tree[4].

In the rest of this section, we discuss briefly the two most widespread algorithms for decision tree induction: C4.5 for classification problems and CART for regression problems.

C4.5 [37] is one of the most popular algorithms for the construction of decision trees. It uses a divide-and-conquer approach to construct a suitable

[4]C5 is a commercially developed variant of C4.5

tree from a set of training examples. Let $\mathcal{S}$ be the set of training examples that reach a node t. The approach will then recursively apply the following steps:

- If $\mathcal{S}$ contains only examples from the class C, then t is a leaf-node and it is assigned the class C.
- If $\mathcal{S}$ contains no examples, then t is a leaf node. Some heuristic must be used to provide a class label to t. C4.5 will assign this leaf the most frequent class at the parent of this node.
- $\mathcal{S}$ contains examples that belong to a mixture of classes. In that case, the algorithm must select a test, based on a single attribute, to split the set $\mathcal{S}$ into smaller subsets $\mathcal{S}_1,\ldots,\mathcal{S}_N$. Each of these subsets will contain all the examples of $\mathcal{S}$ that have the same outcome for this test. The algorithm is then recursively applied to each of the new sets.

One important step is not yet clarified: the selection of the test in case $\mathcal{S}$ contains observations from a mixture of classes. Ideally, we would like the algorithm to find the optimal test, i.e. the test that provides us with the smallest possible tree. Unfortunately, finding the optimal tree is NP-complete and in most cases computationally infeasible. A greedy-heuristic is used to overcome this problem. At each step, we select that attribute and test that produces the 'purest' subsets. An often used measure for the impurity of a node is the entropy. Entropy for the set $\mathcal{S}$ containing observations from C different classes is defined as:

$$Entropy(S) = -\sum_{i=1}^{C} p_i log_2(p_i) \tag{6}$$

with p_i the proportion of examples of class i in the node. We can observe that the entropy serves well as an impurity measure: it is minimal when all observations in $\mathcal{S}$ belong to the same class and maximal when all classes are equally likely. From all possible tests $\mathcal{T}$ that divide the set $\mathcal{S}$ into subsets $\mathcal{S}_1,\ldots,\mathcal{S}_N$, the algorithm will select the test $\mathcal{T}^*$ that maximizes:

$$Gain(\mathcal{S},\mathcal{T}) = Entropy(\mathcal{S}) - \sum_{i=1}^{N} \frac{|\mathcal{S}_i|}{|\mathcal{S}|} Entropy(\mathcal{S}_i) \tag{7}$$

with $|\mathcal{S}|$ representing the number of observations in $\mathcal{S}$. The test $\mathcal{T}^*$ is thus selected such that the weighted entropy in the subsets is minimal. Maximalization of the Information Gain has however one serious deficiency: it favors tests with many outcomes. The Gain Ratio criteria adjusts the bias by using a normalization term

$$Info(\mathcal{S},\mathcal{T}) = -\sum_{i=1}^{N} \frac{|\mathcal{S}_i|}{|\mathcal{S}|} log_2 \frac{|\mathcal{S}_i|}{|\mathcal{S}|} \tag{8}$$

and maximizing the following Gain Ratio

$$GainRatio(\mathcal{S}, \mathcal{T}) = \frac{Gain(\mathcal{S}, \mathcal{T})}{Info(\mathcal{S}, \mathcal{T})} \tag{9}$$

In principle, the recursive splitting can be continued until each leaf node contains only one observation. This strategy would however lead to an overly complex tree with many internal and leaf nodes that overfits the training data. A more parsimonious tree can often be found that provides better generalization performance. Earlier work tried to obtain such a parsimonious tree by not dividing a node if the improvement achieved by the best test was smaller than some predefined threshold [22, 37]. This approach suffers however from the fact that the greedy approach looks only one step ahead. It is possible that the improvement at this step is only small, while the step after it could lead to a substantial improvement. Early stopping is therefore no longer used by most induction algorithms, but replaced by a separate pruning phase. After construction of the largest possible tree, the pruning phase recursively merges leaf nodes until the best possible tree is obtained. C4.5 uses a heuristic based on the binomial distribution to decide whether pruning will improve generalization behavior of the tree or not.

A second very popular tree induction algorithm is **CART**, short for Classification and Regression Trees [9]. We will only discuss the version of CART used for induction of regression trees. The variant for classification trees is largely similar to C4.5, but with a different splitting criterion (Gini Index) and pruning procedure.

A CART regression tree [9] is a binary tree with conditions specified next to each non-leaf node. Classifying a new observation is done by following the path from the root towards a leaf node, choosing the left node when the condition is satisfied and the right node otherwise, and assigning to the observation the value below the leaf node. This value below the leaf nodes equals the average y-value of training observations falling into this leaf node. Variants [28] of CART allow the predictions of the leaf nodes to be linear functions of the input variables.

Similarly to a classification tree, the regression tree is constructed by iteratively splitting nodes, starting from only the root node, so as to minimize an impurity measure. Often, the impurity measure for regression problems of a node t is calculated as:

$$R(t) = \frac{1}{N} \sum_{\mathbf{x}_n \in t} (y_n - \overline{y}(t))^2 \tag{10}$$

with $(\mathbf{x}_n, y_n)$ the training observations and $\overline{y}(t)$ the average y-value for observations falling into node t. The best split for a leaf-node of the tree is chosen such that it minimizes the impurity of the newly created nodes. Mathematically, the best split s* of a node t is that split s which maximizes:

$$\Delta R(s, t) = R(t) - p_L R(t_L) - p_R R(t_R) \tag{11}$$

with t_L and t_R the newly created nodes and p_L and p_R the proportion of examples sent to respectively t_L and t_R. Pruning of the nodes is performed afterwards to improve generalization behavior of the constructed tree. Pruning in CART is performed by a procedure called 'minimal cost complexity pruning' which assumes that there is a cost associated with each leaf-node. We will briefly explain the basics behind this pruning mechanism, more details can be found in [9, 29]. By assigning a cost α to each leaf node, we can consider the total cost of a tree to consist of two terms: the error rate of the tree on the training data and the cost associated with the leaf nodes.

$$\text{Cost(Tree)} = \text{Error(Tree)} + \alpha \text{NumberLeafNodes(Tree)} \tag{12}$$

For different values of α, the algorithm first looks for the tree that has a minimal total cost. For each of these trees, the algorithm estimates the error on a separate data set that was not used for training and remembers the value of α that resulted in the tree with minimal error. A new tree is then constructed from all available data and subsequently pruned based on this optimal value of α. In case there is only limited data available, a slightly more complex cross-validation approach is followed, for which the details are provided in [9].

3.2.3 TREPAN

TREPAN [13, 15] is a popular pedagogical rule extraction algorithm. While it is limited to classification problems, it is able to deal with both continuous and nominal input variables. TREPAN shows many similarities with the more conventional decision-tree algorithms that learn directly from the training observations, but differs in a number of respects.

First, when constructing conventional decision trees, a decreasing number of training observations is available to expand nodes deeper down the tree. TREPAN overcomes this limitation by generating additional instances. More specifically, TREPAN ensures that at least a certain minimum number of observations are considered before assigning a class label or selecting the best split. If fewer instances are available at a particular node, additional instances will be generated until this user-specified threshold is met. The artificial instances must satisfy the constraints associated with each node and are generated by taking into account each feature's marginal distribution. So, instead of taking uniform samples from (part of) the input space, TREPAN first models the marginal distributions and subsequently creates instances according to these distributions while at the same time ensuring that the constraints to reach the node are satisfied. For discrete attributes, the marginal distributions can easily be obtained from the empirical frequency distributions. For continuous attributes, TREPAN uses a kernel density based estimation method [50] that calculates the marginal distribution for attribute x as:

$$f(x) = \frac{1}{m} \sum_{i=1}^{m} \frac{1}{\sqrt{2\pi}\sigma} e^{-(\frac{x-\mu_i}{2\sigma})^2} \tag{13}$$

with m the number of training examples, μ_i the value for this attribute for example i and σ the width of the gaussian kernel. TREPAN sets the value for σ to $1/\sqrt{m}$. One shortcoming of using the marginal distributions is that dependencies between variables are not taken into account. TREPAN tries to overcome this limitation by estimating new models for each node and using only the training examples that reach that particular node. These locally estimated models are able to capture some of the conditional dependencies between the different features. The disadvantage of using local models is that they are based on less data, and might therefore become less reliable. TREPAN handles this trade-off by performing a statistical test to decide whether or not a local model is used for a node. If the locally estimated distribution and the estimated distribution at the parent are significantly different, then TREPAN uses the local distributions, otherwise it uses the distributions of the parent.

Second, most decision tree algorithms, e.g. CART [9] and C4.5 [37], use the internal (non-leaf) nodes to partition the input space based on one simple feature. Trepan on the other hand, uses M-of-N expressions in its splits that allow multiple features to appear in one split. Remember from Section 1.3 that an M-of-N split is is satisfied when M of the N conditions are satisfied. 2-of-{a,¬b,c} is therefore logically equivalent to (a $\wedge$ ¬b) $\vee$(a $\wedge$ c) $\vee$(¬b $\wedge$ c). To avoid to test all of the possible large number of M-of-N combinations, TREPAN uses a heuristic beam search with a beam width of 2 to select its splits. The search process is initialized by first selecting the best binary split at a given node based on the information gain criteria([13] (or gain ratio according to [15]). This split and its complement are then used as basis for the beam search procedure that is halted when the beam remains unchanged during an iteration. During each iteration, the following two operators are applied to the current splits:

- m-of-n+1: the threshold remains the same but a new literal is added to the current set. For example, 2-of-{a,b} is converted into 2-of-{a,b,c}
- m+1-of-n+1: the threshold is incremented by one and a new literal is added to the current set. For example, 2-of-{a,b} is converted into 3-of-{a,b,c}

Thirdly, while most algorithms grow decision trees in a depth-first manner, TREPAN employs the best-first principle. Expansion of a node occurs first for those nodes that have the greatest potential to increase the fidelity of the tree to the network.

A final difference between TREPAN and more conventional decision tree algorithms concerns the stopping criteria used by TREPAN to decide when to stop growing the tree. Conventional algorithms will first construct a complete tree and prune this tree afterwards. TREPAN on the other hand uses both local and global criteria to decide on the optimal tree. Local criteria only take the current node into account, such as the number of training instances or their class distribution, to decide whether or not to expand the current node. The local criterion uses by TREPAN is based on the purity of the tree. A node will become a leaf if, with a high probability, it only covers examples from a single

class. TREPAN will create a confidence interval and calculate the number of instances necessary to be able to decide whether $P(prop_c < 1 - \epsilon) < \alpha$ with α a significance level and ϵ an indication of how tight the interval must be.

Besides such a local criterion TREPAN also employs global criteria. Unlike local criteria, global criteria take the entire tree into account and not only the node currently considered for expansion. TREPAN allows the user to specify a complexity parameter, a maximum number of internal nodes, as global criterion to limit the size of the tree returned by TREPAN. Additionally, a validation data set can be provided. TREPAN uses this set to measure the fidelity of each tree created and returns the tree with the highest fidelity.

3.2.4 ANN-DT and DecText

The **ANN-DT** (Artificial Neural Network Decision Tree) algorithm is described by its authors as 'an algorithm that extracts binary decision trees from a trained neural network' [42]. Although the name of the algorithm suggests a close intertwining of the extraction algorithm with the underlying neural network, ANN-DT does not place any restrictions on the structure of this underlying model and is therefore also applicable with other types of black box classifiers. Additionally, ANN-DT can be applied to data sets where both inputs and outputs can be discrete or continuous. In the rest of this section, we discuss the variant of the algorithm developed for regression problems [42].

ANN-DT shares the idea of TREPAN to sample the input space to create additional training instances. The sampling method used is however different from TREPAN's that was based on the features' marginal distributions. ANN-DT randomly generates instances but only keeps those instances for which the distance towards the nearest original training observation is smaller than some predefined critical value. This ensures that only data points are created that resemble the original training data. The critical distance is determined by selecting a large enough sample of the training data and taking the average distance between these points and their 10 nearest neighbors. The new instances are then presented to the 'black box', from which a predicted output is obtained.

Afterwards, a binary decision tree is constructed from these artificial instances by recursively splitting the input space based on one of the input attributes. Two different variants of ANN-DT exist that differ in the selection method of the best attribute to split on and an appropriate threshold. The first variant, ANN-DT(e), makes these choices such that the weighted variance in the newly created branches is minimized. This is the same criterion as used in CART's regression trees. The second variant, ANN-DT(s), is computationally more expensive and selects as splitting attribute the feature that has most significance on the behavior of the underlying model. The mathematical details can be found in [42], but the idea behind this type of feature selection is to find the attribute a that has the largest significance $\sigma(f)_a$ for the black box function f($\mathbf{x}$) at the current branch. For significant attributes, changes in the attribute's value will be related to variation in the function.

Whatever the method used to select the best splits, ANN-DT recursively

divides the current input space in two subspaces. Growing of the tree is terminated when the variance in a node becomes zero or when one of the stopping criteria is satisfied. A first (local) stopping criteria is to expand a node only when the mean outputs of the instances in each of the two children are significantly different from each other. This is tested with an F-test for all nodes below a user-specified minimum depth in the tree. Similarly to TREPAN, users can also specify a global complexity parameter: the maximum depth of the tree. Nodes at this level will not be further expanded.

Empirical studies conclude that ANN-DT had similar or better performance than those obtained from CART. The ANN-DTT(s) variant with significance analysis to select the splitting attribute, was found to provide more accurate rules.

DecText[8], short for Decision Tree Extractor, is very similar to TREPAN and ANN-DT. It was designed specifically for the extraction of a classification decision tree from a neural network and to obtain this goal several different splitting criteria were proposed. We will briefly discuss one of these criteria, SetZero, and the pruning mechanism.

The SetZero splitting method tries to find the attribute that has most effect on the outputs of the underlying model. This is similar to the idea behind ANN-DT(s) but differs in how the split is actually calculated. First, for all of the observations that fall in a specific node, the underlying model is queried to find the corresponding outputs. Then, the algorithm places the value of the first attribute to zero in all the observations and queries the network again to find its predictions for these mutated inputs. This process is repeated for each of the possible attributes and the absolute difference between the original outputs and the mutated outputs indicates how much the outcome is affected by the attribute. The algorithm will select the attribute with the largest influence. One can observe that the SetZero splitting mechanism can only be applied if there is an underlying model that can provide predictions for the mutated observations. It can therefore not be applied directly on training data.

Besides the special splitting criterion, DecText also uses a special pruning mechanism, called Fidelity Pruning. First, a number of random observations is created and the corresponding output is obtained from the black box. Pruning of leaf nodes is then performed as long as it improves the fidelity on this random data set.

3.2.5 GEX and G-REX

Various approaches based on the ubiquitous genetic algorithms have been proposed to tackle the problem of rule extraction. In this section, we will discuss GEX [31] and G-REX [25], (both abbreviations for 'Genetic Rule EXtraction') as examples of this approach. In Figure 7, the basic idea behind genetic algorithms is explained.

First, a random population of individuals is created. Each individual represents a possible solution to the proposed problem. For rule extraction, an

```
1  Initialize a population of individuals
2  While (stopping criteria not met)
3     Evaluate fitness of each individual and rank them accordingly
4     Apply genetic operators on these individuals
5     Update population
6  End While
```

Figure 7: The Basic Genetic Algorithm

individual will therefore correspond with a rule or a rule set. Afterwards, the algorithm will calculate a level of fitness for each individual. This problem-specific fitness function is used to measure how well an individual is able to solve the corresponding problem. For rule extraction, the fitness measure will incorporate elements such as accuracy and comprehensibility of the rule (set). Consequently, two individuals will be selected to create offspring whereby individuals that are fitter than others will have a greater probability of being selected. Their children will inherit elements from both parents and form a new and updated population. Some of the parents will also be added directly to this updated population: either unchanged or with slight mutations. The general idea is Darwin's survival of the fittest: small mutations and inheritance will give increasingly better solutions to solve the original problem. The algorithm will stop when a maximum number of iterations is reached or when an individual is found that is sufficiently fit.

GEX [31] follows the general approach of a genetic algorithm, but differs on several aspects. The first difference is that GEX does not work with one population but with several subpopulations evolving on islands. There are as many islands as there are classes and each subpopulation is specialized in searching rules for one specific class.

The individuals in these populations represent rules. To allow rules with different lengths, a special encoding is used for the representation of these rules (Figure 8). This fixed length representation, called a chromosome, contains a condition part and a conclusion part. The condition part is subdivided in as many genes as there are input variables. The binary flag in each of these genes indicates whether the conditions on the associated input variable are active or not. When the flag is set to zero, the conditions in the rest of the gene do not apply. The encoding of these conditions is dependent on the type of the input variable the gene is associated with. For example, when the input variable is a binary variable, the rest of the gene is just a zero or one, but if the associated variable is continuous then the rest of the gene consists of an operator and two threshold values X_1 and X_2. The gene can then represent different conditions such as $x < X_1$ or $X_1 < x < X_2$ depending on the value of the operator part. The gene of the conclusion indicates the class.

GEX uses three kinds of genetic operators: mutation, crossover and mi-

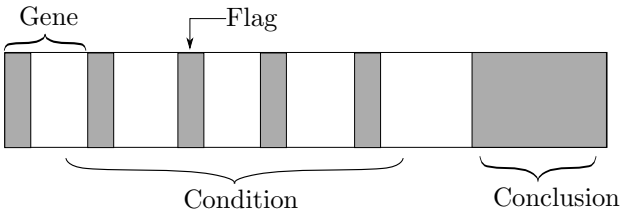


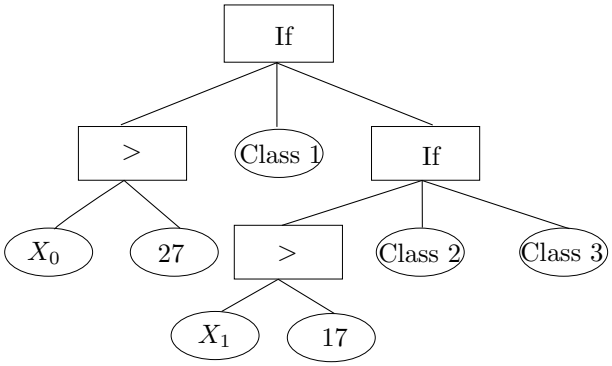
Figure 8: Rule Representation in GEX

gration. Mutation is applied on the chromosomes and randomly changes the information encoded in the genes. Crossover is a gene exchange between two chromosomes and results in two new individuals. Migration is a special operator that transfers weak individuals to a different island. The fitness function to evaluate the rules is based on a weighted average of several aspects such as accuracy, coverage and comprehensibility. By specifying the weights, the analyst can indicate the importance of each of these criteria. Finally, GEX removes those rules that are more specific than other rules or that do not satisfy certain criteria, such as a minimum accuracy or coverage before starting a new iteration.

One of the main drawbacks of GEX is the fact that one observation can be covered by multiple rules. Multiple rules can be true for a certain instance and it also not guaranteed that at least one rule will be valid.

The second genetic rule extraction algorithm that is discussed, **G-REX** [25, 26] overcomes these problems and delivers exclusive and exhaustive rules. It is also not limited to classification problems. Furthermore, G-REX can produce different types of rules, e.g. decision trees or fuzzy rules. G-REX uses a different representation than GEX to allow the encoding of rules of various length. G-REX is based on a subbranch of genetic algorithms that is called genetic programming and uses S-expressions to represent the individuals. A sample S-expression and its corresponding tree representation are given in Figure 9. Each S-expression contains elements from two sets: a function set and a terminal set. Functions are the internal nodes of the tree and represent operations that can be applied on the terminals. The analyst must define these operations and supply the number of arguments that each function can take. The terminals, the leaf nodes in the tree, are the variables and constants from the problem. In the example expression, there are two functions ('if' and '>') and 7 terminals (X_0, X_1, two constants and three class labels).

S-expressions are a powerful alternative to GEX's flag-mechanism for the encoding of rules or rulesets with varying length. For example, the S-expression of Figure 9 is equivalent to the following ruleset: 'if X_0 is larger than 27 then classify as Class 1, else classify as Class 2 if X_1 is larger than 27 and as Class 3 otherwise'. Whereas the above expression is equivalent to a simple decision tree, changing the elements of the terminal and function sets allows different



(if ($> X_0$ 17) Class1 (if($> X_1$ 27) Class2 Class3))

Figure 9: S-Expression and its corresponding Tree Representation

types of rules to be created. The fitness function used to select the fittest individuals is very similar to the one used by GEX and incorporates elements, such as comprehensibility and accuracy.

In [38], an approach was proposed that is very similar to G-REX. The main difference is that artificial examples are used to evaluate the fitness of the rules. The algorithm is outlined in Figure 10. First the set $\mathcal{S}$ is initialized: the training observations with the outputs provided by the underlying black box are inserted in $\mathcal{S}$ and an initial population of rules is created. The algorithm will then create a user-specified number of artificial examples that are also inserted into the set $\mathcal{S}$. Creation of the artificial examples is performed by taking a training observation and mutating random attribute values. Before inserting the example into $\mathcal{S}$, a check is performed to see if the example is not already present in $\mathcal{S}$ and to verify whether the output of the black box for this example is different from the output of the current best rule. If the example passes these checks, it is added into $\mathcal{S}$. The genetic algorithm will then be executed several times. After this, the correctly classified examples, i.e. examples for which the best rule prediction equals the black box prediction, are removed from $\mathcal{S}$ and the entire process is repeated until the algorithm finds rules that are fit enough.

The wide variety of genetic rule extraction algorithms proves that they are a useful tool for the analysis of black box models. The main advantage of these methods is their flexibility to changes. Altering the fitness function allows the analyst to select the required accuracy-comprehensibility trade-off and changing the function set even allows the user to change the format of the extracted descriptions. The main drawback of all genetic algorithms are the computational requirements for performing the successive iterations. A second drawback concerns the consistency of the extracted descriptions. Due to the aspect of probability during creation of the rules, we can expect the extracted rules to be significantly different when a genetic algorithm is run several times

```
1  Initialize the set S
2  Create initial population of rules R
3  While (rules in R are not good enough)
4     Create Examples and Add them to S
5     Evaluate fitness of rules in R
6     Apply genetic programming for N iterations
7     Remove from S the examples for which the output
8        of the ANN equals the output of the best rule
9  End While
```

Figure 10: Algorithm of Rabuñal et al. [38]

on the same data set.

3.2.6 BIO-RE

One of the most straightforward rule extraction algorithms is BIO-RE (Binarized Input-Output Rule Extraction) [51]. As suggested by its name, the method is only applicable when all inputs and outputs are binary variables. For other types of variables, a transformation is required. For numerical variables, it is suggested to use the binarization method of Equation 14.

$$x_i' = \begin{cases} 1 & \text{if } x_i \geq \mu_i; \\ 0 & \text{otherwise.} \end{cases} \tag{14}$$

with μ_i the mean value over all x_i's. The idea behind BIO-RE is to create each possible input combination and to query the network for the corresponding output decision. A truth table is generated from the samples and boolean simplification methods are subsequently used to convert this truth table into the corresponding boolean function.

One can observe immediately the disadvantages of this method: because the algorithm first creates all possible input combinations, the method is only applicable when the number of inputs N is small as the number of combinations equals 2^N. For many real-word problems this number of combinations is very large, making the method computationally infeasible. The required binarization is also undesirable as it will probably deteriorate classification performance. Because of these two shortcomings, the method seems not applicable to any real-life problem.

3.2.7 BUR

In [10], a pedagogical algorithm for rule extraction, called Boost Unordered Rule Learner (BUR), was proposed. Although BUR was proposed as a rule extraction algorithm for SVM classifiers, it can also be used to learn rules directly form the data points. BUR creates non-exclusive propositional rules and associates with each rule a weight or confidence level. New observations are

classified by finding all rules applicable and combining them such that rules with higher confidence are more important for the final classification decision. BUR incorporates elements from both the rule learner CN2 [11] and 'gradient boosting machines'[19]. We will first provide a detailed explanation of the 'gradient boosting mechanism' and afterwards explain how it can be combined with CN2 to perform rule extraction.

Gradient Boosting Machines

In [19], Friedman proposes a method for function estimation coined 'Gradient Boosting Machines'. Given a set of training observations $\{\mathbf{x}_i, y_i\}_1^N$, the goal of function estimation is to find a function F*($\mathbf{x}$) that minimizes a user-defined loss function $\psi(y, F(\mathbf{x}))$. For regression problems, the most widespread loss functions are the squared error and absolute error function.

In [19], Friedman limits the allowed format of the functions F($\mathbf{x}$) to:

$$F(\mathbf{x}, \mathbf{P}) = \sum_{m=0}^{M} \beta_m h(\mathbf{x}; \mathbf{a}_m) \tag{15}$$

with $\mathbf{P} = \{\beta_m, \mathbf{a}_m\}_0^M$ a set of parameters. The functions $h(\mathbf{x}, \mathbf{a}_m)$ are called base learners and are usually chosen to be simple functions of $\mathbf{x}$ with parameters $\mathbf{a}$. Possible base learners are single propositional rules, decision trees or linear functions. Friedman refers to several other function approximation methods, such as neural networks and Support Vector Machines, that share the format of Equation 15.

The problem of function estimation can then be formulated as finding the parameters P for which the loss function is minimized:

$$\{\beta_m, \mathbf{a}_m\}_{m=0}^M = arg \min_{\beta_m', \mathbf{a}_m'} \sum_{i=1}^{N} \psi(y_i, \sum_{m=0}^{M} \beta_m' h(\mathbf{x}; \mathbf{a}_m')) \tag{16}$$

Depending on the loss function and format of the base learners, it might be infeasible to find a solution to this problem and a greedy approach might be required. Starting from an initial approximation F_0, the algorithm iteratively looks for the best step towards the optimal solution given the current approximation. For m=1,...,M do:

$$\{\beta_m, \mathbf{a}_m\} = arg \min_{\beta, \mathbf{a}} \sum_{i=1}^{N} \psi(y_i, F_{m-1}(\mathbf{x}_i) + \beta h(\mathbf{x}_i; \mathbf{a})) \tag{17}$$

$$F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \beta_m h(\mathbf{x}; \mathbf{a}_m) \tag{18}$$

For some loss functions and/or base learners it might be difficult to solve Equation 17 and in those cases gradient descent can be used to find an approximate solution.(Appendix A)

Assuming $F_{m-1}(\mathbf{x})$ is the approximation during the m-th iteration, gradient descent searches for a $F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) - \epsilon g_{m-1}(\mathbf{x})$ with

```
    Initialize F_0
    For m=1 to M do
1     g_{m-1}(x) = ∂ψ(y,F_{m-1}(x))/∂F_{m-1}(x) |_{x=x_i}    i = 1,...,N
2     a_m = arg min_{a,β} Σ_{i=1}^{N} (−g_{m-1}(x_i) − βh(x_i;a))^2
3     β_m = arg min_β Σ_{i=1}^{N} ψ(y_i, F_{m-1}(x_i) + βh(x_i;a_m))
4     F_m(x) = F_{m-1}(x) + β_m h(x;a_m)
    End For
```

Figure 11: Gradient Boosting

$$g_{m-1}(\mathbf{x}) = \frac{\partial \psi(y, F_{m-1}(\mathbf{x}))}{\partial F_{m-1}(\mathbf{x})} \tag{19}$$

If $g_{m-1}(\mathbf{x})$ has the same format as the base learners, and in that case can also be written as $h(x, \mathbf{a}_m)$, then we can find a value for β_m by minimizing:

$$\beta_m = arg \min_{\beta} \sum_{i=1}^{N} \psi(y_i, F_{m-1}(\mathbf{x}_i) + \beta h(\mathbf{x}_i; \mathbf{a}_m)) \tag{20}$$

However, $g_{m-1}(\mathbf{x})$ will usually have a different format than the base learners, and in that case we will take as $h(x, \mathbf{a}_m)$, the $h(x, \mathbf{a})$ that is most correlated with $g_{m-1}(\mathbf{x})$ over the training data. Therefore, $\mathbf{a}_m$ is the solution of:

$$\mathbf{a}_m = arg \min_{\mathbf{a},\beta} \sum_{i=1}^{N} (-g_{m-1}(\mathbf{x}_i) - \beta h(\mathbf{x}_i; \mathbf{a}))^2 \tag{21}$$

A value for β_m can again be obtained from Equation 20. A general overview of the gradient boosting mechanism is given in Figure 11.

Integration with CN2

BUR is based on the above concept of 'gradient boosting' and a single propositional rule is chosen as the format of each base learner. Formally,

$$r(\mathbf{x}) = h(\mathbf{x}, \mathbf{a}) = \begin{cases} +1/-1 & \text{if the rule is applicable on example } \mathbf{x} \\ & \text{and classifies examples as positive/negative} \\ 0 & \text{otherwise.} \end{cases} \tag{22}$$

An overview of the basic algorithm is shown in Figure 12. We will explain the differences with Figure 11 below.

The loss function used by BUR is the absolute error function. The calculation of $g_m(\mathbf{x})$ (Step 1 in Figure 11) can therefore be simplified to:

	Initialize F_0
	For m=1 to M do
1	$g_{m-1}(\mathbf{x}_i) = -sign(y_i - F_{m-1}(\mathbf{x}_i)), \quad i = 1, \ldots, N$
2a	$X = X - \{\mathbf{x}_i \vert g_{m-1}(\mathbf{x}_i) = 0\}$
2b	$Y = \{g_{m-1}(\mathbf{x}_i) \vert \mathbf{x}_i \in X\}$
2c	$r_m = LEARN - ONE - RULE(X, Y)$
3	$\beta_m = \text{median}\{\frac{y_i - F_{m-1}(\mathbf{x}_i)}{r_m(\mathbf{x}_i)} : \text{all } \mathbf{x}_i \text{ covered by rule } r_m\}$
4	$F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \beta_m h(\mathbf{x}; \mathbf{a}_m)$
	End For

Figure 12: Boost Rule learner

$$g_{m-1}(\mathbf{x}) = \frac{\partial \psi(y, F_{m-1}(\mathbf{x}))}{\partial F_{m-1}(\mathbf{x})} = -sign(y - F_{m-1}(\mathbf{x}))^5 \tag{23}$$

Step 2, finding the best base learner, is the place where the integration with CN2 occurs. In step 2a, the algorithm checks which points are correctly classified and for these points that are currently misclassified, the algorithm looks for a rule that minimizes the current approximation's error.

Due to the format of the base learners and the absolute error function, the third step can also be simplified to:

$$\beta_m = \text{median}\{\frac{y_i - F_{m-1}(\mathbf{x}_i)}{r_m(\mathbf{x}_i)} : \text{all } \mathbf{x}_i \text{ covered by rule } r_m\} \tag{24}$$

The complete algorithm of BUR is a slightly more advanced version of the one described in Figure 12. To overcome some problems explained in [10], rules are learned for each class in turn. There are also some changes in step 2a-2c: it will remove the training examples from the current class that are correctly classified by the approximation function with a confidence that is at least as large as the corresponding SVM and learn rules from the remaining examples. The complete BUR algorithm with these minor improvements can be found in [10].

After the rule learning phase, pruning of the returned rule set is done by a greedy strategy. Until the number of rules in the rule set is smaller than some user-specified threshold, the pruning algorithm will remove in each step the rule that has the smallest influence on the overall performance.

While experiments show that the rules extracted by BUR are accurate, we believe that the main disadvantage of this technique lies in the complexity of the rules that are generated. Whereas each rule on its own is relatively easy to

[5]The author of [10] uses a sign function with three possible outcomes instead of the more widespread binary sign-function: $sign(x) = 1$ for $x > 0$, $sign(0) = 0$ and $sign(x) = -1$ for $x < 0$

understand, the mechanism for combining the individual predictions requires a summation of the confidence factors of all applicable rules. This combination mechanism degrades the comprehensibility of the extracted description.

3.2.8 REFNE and STARE

In [61], REFNE is presented as an algorithm for classification **R**ule **E**xtraction **F**rom **N**eural network **E**nsembles. While the name suggests a close intertwining between the algorithm and underlying ensemble, the algorithm is pedagogical as it only uses this ensemble as an oracle. The rule set extracted by REFNE consists of ordered propositional rules. Rules extracted first have a higher priority and act as implicit antecedents of the ones extracted later. To improve comprehensibility, the algorithm ensures that the maximum number of antecedents in the condition part of each rule is limited to three. While this limitation can result in a larger number of rules, the authors believe that the smaller size of each rule improves comprehensibility. Additionally, REFNE prefers categorical attributes as antecedents. Only if no suitable rule can be found based on the categorical attributes, REFNE considers the use of numerical attributes for insertion in the condition part. REFNE is in many aspects similar to STARE [60] that was proposed by the same authors to extract rules from a single neural network.

The basics behind REFNE are relatively easy to explain. Just like TREPAN and ANN-DT, REFNE starts with the generation of artificial data examples. Unlike TREPAN and ANN-DT, the sampling process selects completely random instances in the value ranges of the input attributes. Once this instance set $\mathcal{S}$ is created, REFNE starts extracting a rule set $\mathcal{R}$. Rule creation is a straightforward breadth-first search. A random (categorical) attribute a_i is picked and for each possible value u of this attribute it is checked whether all instances of $\mathcal{S}$ that have this value for the attribute belong to the same class. If this is the case a new candidate rule is created, and otherwise another categorical attribute is selected and the process repeats.

If no rule could be created after checking all single attributes, the algorithm will try to create rules with two conjunctive antecedents. If this is also unsuccessful, rules with three antecedents are being looked for. Failure to find a rule after these steps, will lead to the discretization of a continuous attribute. The above process will then be repeated over all categorical attributes and the newly created discretized continuous attribute. When a candidate rule r is found, its fidelity is evaluated on a newly created sample and, if above a user-specified threshold, r is added to the rule set $\mathcal{R}$. Instances covered by r are removed from $\mathcal{S}$ and the entire process is iterated until $\mathcal{S}$ is empty or until no suitable rule was found.

The choice of this fidelity threshold seems therefore rather important. Selecting a value that is too large will result in frequent rejection of candidate rules. It is therefore possible that not enough rules are found to obtain a rule base that covers all possible inputs: exhaustiveness is not guaranteed. Setting the threshold value too low results in the opposite situation: each candidate

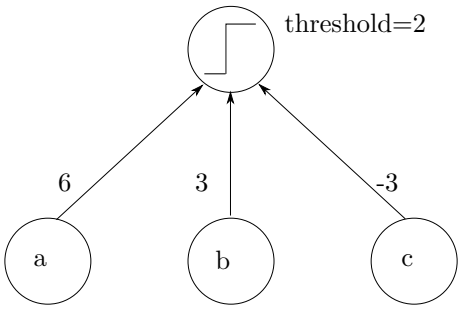


Figure 13: Subset (a,b,c= boolean inputs)

rule is accepted, even rules with low accuracy.

3.2.9 ITER

In [23], we propose an independent algorithm for regression rule extraction: ITER. The method works by an iterative expansion of hypercubes, whereby each hypercube represents a propositional rule. The search process corresponds to a specific-to-general search and ensures that the extracted rules are both exclusive and exhaustive. With minor changes, ITER is also able to extract rules for classification problems. For more details on this algorithm, we refer to [23].

3.3 Dependent Algorithms

All the algorithms discussed in the previous section are completely independent of the underlying model. The black box can therefore be replaced with a different model without affecting the extraction algorithm. The algorithms in this section depart from this independence: they require specific models and use their knowledge of the underlying models during the extraction process. We will see that most of these dependent algorithms are specifically designed for neural network rule extraction, with only a few exceptions developed for other models.

3.3.1 SUBSET

One of the first neural network decompositional algorithms was the Subset method by Towell and Shavlik [56], which is however very similar to the KT-method of Fu [20], PARTIAL-RE [51] and the approach by Saito and Nakano [39].

The basic idea behind all of these approaches is to find combinations (subsets) of inputs that guarantee activation of the output unit irrelevant of the values of the other units. Usually a breadth-first search is performed to find

these combinations. We will show this with the example of Figure 13 (based on [14]). In this example, three binary inputs are connected to a single output unit with the weights that are shown in the figure. We are looking for rules that guarantee activation of the output unit. First, all rules with exactly one antecedent are evaluated. From these 6 rules: (a,$\neg a$,b,$\neg b$, c,$\neg c$), only one will always ensure activation of the output unit, namely the rule having the antecedent a. This can be checked by taking values for b and c that least support activation of the output unit. Unit b is connected with a positive weight to the output unit. This unit should therefore take the value of false (i.e. 0) to be minimally supportive and similarly for unit c we can deduce that this unit should take the value true (or 1) to be minimally supportive. The input to the output unit is then equal to (6)(a)+(3)($\neg$b)+(-3)(c)=3 which exceeds the threshold for this unit. Thus, whatever the choice for nodes b and c, the output unit will be active when node a is active: a first rule is found 'if a=1 then output=1'.

The breadth-first search will then check the rules that have two antecedents and that are not more specific than already found rules. In this example, there are eight such rules to consider with as antecedents: $\neg a \wedge$ b,$\neg a \wedge$ c, $\neg a \wedge \neg b$, $\neg a \wedge \neg c$, b $\wedge$ c, b $\wedge \neg c$, $\neg b \wedge$ c,$\neg b \wedge \neg c$. Only the rule 'if b $\wedge \neg c$ then output=1' will be retained and the breadth-first algorithm will look for possible rules having three antecedents. No such rules are found and the algorithm will terminate.

For multi-layered neural networks, the above search process will be executed for each hidden and output unit. A rewriting procedure is then performed afterwards to eliminate the hidden concepts and to create rules that directly link the input variables to the output.

The above toy-problem was easy to solve. For larger real-life problems the search involved will however become too expensive and impossibly to solve. To face these problems, limitations on the search process must be imposed. In [39, 51], the search is restricted to a certain depth, such that only rules with a maximum number of antecedents can be found. Fu [20] employs information about the weights and activation function to reduce the size of the search space. For example, rules containing $\neg b$ must not be considered for the example above because unit b is connected with a positive weight to the output unit. Whenever a rule is found containing $\neg b$, the same rule with $\neg b$ replaced by b will also exceed the threshold. In that case, the more general rule without the condition $\neg b$ should already be found by the search algorithm. Various other optimizations to facilitate the search process have been proposed: PARTIAL-RE [51] uses weight ordering to efficiently retrieve the candidate subsets, while FULL-RE [51] employs linear programming to find suitable subsets.

The SUBSET algorithm, shown in Figure 14, also restricts the search space by allowing the user to specify a maximum number of rules it may return. Afterwards, a Branch-and-Bound algorithm is used to find the subsets.

After analysis of the rules extracted by SUBSET on several data sets, Towell and Shavlik [56] noticed that the rule set could often be significantly reduced when converted into M-of-N rules. This conclusion led to the development of an algorithm specifically dedicated to the extraction of this type of rules.

```
For each hidden and output unit
  Find up to β_p subsets of positively-weighted income links whose summed weight
  is greater than the bias on the unit
  For each element P of the β_p subsets:
    Extract up to β_n subsets of negatively weighted links whose
    summed weight is greater than the sum of P less the bias on the unit
    With each element N of the β_n subsets form a rule:
    'if P and not N then "name of unit"'
  End For
End For
```

Figure 14: Subset Algorithm [56]

```
For each hidden and output unit
  • Cluster similarly-weighted links and set the weight value of
    each member to the cluster average
  • Eliminate clusters that do not affect activation of the output
  • Use the backpropagation algorithm to optimize bias of the units,
    while keeping the weights fixed to the cluster average
  • Form a single rule for each unit
  • If possible, simplify the rule to an M-of-N rule
End For
```

Figure 15: N-of-M [56]

3.3.2 N-of-M

N-of-M [56] is a decompositional method that analyzes weights of the neural network to find M-of-N rules. M-of-N rules are particularly attractive to describe the behavior of a unit when there are several weights with similar values connected to the unit. For example, assume that a unit with threshold value of 10 is connected to three inputs with a weight of 6. Describing the activation conditions for this unit with propositional rules requires an enumeration of all possible scenarios, {a $\wedge$ b,a $\wedge$ c, b $\wedge$ c}, while only one M-of-N rule is sufficient: 2-of-{a,b,c}.

The algorithm for extraction of M-of-N rules is given in Figure 15. In the first step, weights are clustered into a limited number of clusters and replaced by the average weight over all members of the group. This ensures that the algorithm does not have to consider individual links, but can work with groups of links. In the second step, groups with small weights and few members are pruned from the network as they have no influence on the activation of the output unit. After removal of these irrelevant groups, a restricted version of backpropagation is run to optimize the bias of each unit. Connection weights are not altered and remain fixed at the average of the cluster they belong to. After optimization of

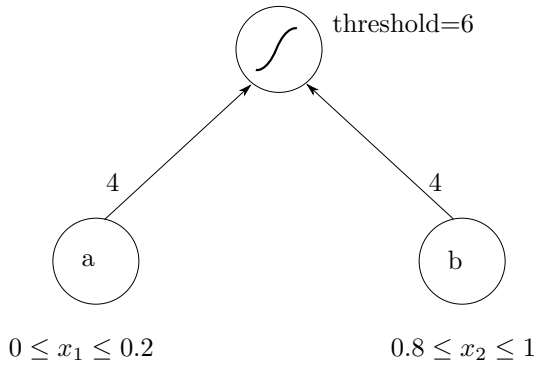


Figure 16: Example of VIA method

the thresholds, rules are extracted for each unit. Exact details of how this is done are not given in the paper, but a SUBSET-like approach might be feasible because the clustering and pruning of irrelevant groups drastically reduces the search tree. For the example of above with three weights of 6 and a threshold of 10, the extracted rule will look like 'IF 6.NumberTrue(a,b,c)>10 THEN active' with NumberTrue() a function that returns how many of its arguments are true. In the final step, the algorithm tries to convert these rules into equivalent M-of-N rules. For the above example this is relatively straightforward: dividing by 6 converts the condition of the rule into 'IF NumberTrue(a,b,c)>1.66' which is similar to at-least 2-of-{a,b,c}.

3.3.3 VIA

Before VIA (Validity Interval Analysis) [52, 53] was developed in 1993, most techniques were just slightly altered variants of the SUBSET method. VIA was one of the first methods that used a totally different approach. However, just like the SUBSET algorithms, VIA is only applicable when the underlying model is a neural network because it analyzes the weights of the network during the extraction process. VIA does not place restrictive assumptions on the architecture or the training of this network. While the algorithm was proposed for classification, we believe it to be easily extensible to regression problems as well.

VIA employs a generate-and-test procedure in its rule extraction process. In the first step candidate rules are generated and these candidate rules are subsequently checked for consistency with the neural network. We will first discuss this second step, checking whether a given rule is consistent with the network, from which the method derives its name.

The principal idea behind this consistency checking is based on the propagation of validity intervals throughout the network. These validity intervals are restrictions on the activation values that each unit in the network can take. We will show how these validity intervals can be propagated through the network

with an example derived from [52] and shown in Figure 16. This simple network might serve as an implementation of the AND-operator. Assume that the inputs A and B are restricted to respectively the intervals [0, 0.2] and [0.8, 1]. The minimum (and maximum) net input to the third unit can then be found by solving a simple linear problem with following restrictions:

$$\begin{aligned} & \min \text{ (or max) } Net_C \\ \\ s.t. \quad & Net_C = 4x_1 + 4x_2 - 6 \\ & x_1 \leq 0.2 \\ & 0 \leq x_1 \\ & x_2 \leq 1 \\ & 0.8 \leq x_2 \end{aligned} \tag{25}$$

Solving this problem will provide us a validity interval for the third unit: $Net_C \in$ [-2.8,-1.2], which can be converted to an output interval by applying the transfer function of this unit. If we assume a sigmoid function then $X_3 \in$ [0.05732, 0.2314]. In this example we propagated the validity intervals forwards through the network, but a similar approach can be followed to propagate validity intervals backwards through the network. In each propagation step, a newly derived bound will replace an older bound only when the new interval becomes smaller. Thus, starting from a number of initial validity intervals every iteration either refines a validity interval or keeps it stable. After a finite number of iterations, the process will therefore converge to a stable situation where no more refinements are possible. The more interesting situation however occurs if during the refinements an empty interval is obtained for one of the validity intervals. Obtaining such an empty interval means that the initial intervals were inconsistent: it is not possible to obtain a combination of activation patterns that satisfy all initial constraints.

Testing the correctness of a conjectured rule can then be performed by negating the rule, using this negated rule to specify the initial intervals and propagating these intervals through the network. If this propagation results in an inconsistency, it is a proof that the opposite rule is false, which implies correctness of the original rule. If the propagation does not give an inconsistency, nothing can be concluded about the original rule. This is caused by the fact that VI-analysis is not able to detect all inconsistencies. The refinement process considers all inputs to a node as independent of each other, but this is not the case when the network consists of multiple layers. VI-analysis is therefore overly conservative in its refining of the intervals. Failure of finding an inconsistency might therefore be caused by either an incorrect original rule or just by the inability of the VI-analysis to find the inconsistency. Therefore, nothing can be concluded about the original rule when the algorithm converges normally.

One final aspect remains to be discussed: How does VIA generates rules that can subsequently be tested for correctness. Two different rule-generation methods were proposed depending on the type of inputs.

If the inputs are discrete, then a search is performed that is very similar to the breadth-first search of the SUBSET type of algorithms. First, all rules

with one antecedent are tested and if proven correct added to the rule base. Afterwards, rules with an additional antecedent, and that are not more specific than an already found rule, are provided as input to the VI-analysis. Such generation process is fast when simple rules cover most of the input space but becomes slow otherwise.

For continuous inputs, the above rule generation process is infeasible. Therefore a second rule generation process, specifically designed for continuous inputs was proposed. One starts from a single data point and its class label as predicted by the network. This single data point already forms a rule and VI-analysis, with validity intervals set to single points, can easily prove this rule to be correct. To find more general rules, one of the inputs is randomly chosen and a small value is added to the bounds of its validity interval. VI-analysis is then performed to check whether the generalized rules can still proven to be correct. This process will continue until enlarging each of the boundaries, results in a rule that can no longer be proven correct. As shown in [23], this iterative growing of rules is very similar to the approach of the ITER-algorithm. There are however differences between the two methods. Whereas the above method randomly selects an input, ITER will use a similarity criterion to find the most appropriate interval to enlarge. Additionally, ITER ensures that extracted rules are non-overlapping and does not rely on interval analysis to decide whether enlargement of an interval is appropriate.

3.3.4 NeuroLinear

The decompositional technique NeuroLinear [48] is able to extract oblique classification rules from neural networks with one hidden layer. The extraction process proceeds as follows:

Step 1 Train and prune a neural network. A penalty term is added to the network error function to ensure small weights for irrelevant units. A pruning algorithm is then applied to remove these irrelevant connections.

This step is essential for the creation of a network that generalizes well and to allow the extraction of a small set of comprehensible rules.

Step 2 Discretize the hidden unit activation values. For each hidden unit, the possible activation values are grouped into a few clusters. An algorithm, called Chi2 [46], was developed to perform this clustering and it ensures that the accuracy of the network is preserved as much as possible.

step 3 Extraction of Rules. The actual process of rule extraction occurs in three phases. First, rules are created that describe the class predictions in terms of the discretized hidden unit activation values. Second, rules are extracted that describe the discretized hidden unit activation values in function of the original attributes. Finally, the rules from the previous steps are combined to find rules that predict the class labels in function of the original attributes.

For the first phase, the authors propose the use of the rule generator X2R [47]. It is capable of learning rules with a specified level of accuracy from a number of examples, each consisting of a set of discretized activation values together with the corresponding class label. The result of this step is a set of rules, each describing the class prediction in function of the discretized hidden activation values.

The second phase, expressing the discretized activation values in terms of the original attributes, is straightforward. Let the hidden unit activation value for neuron j be H_j. This value will fall within one of the K clusters found by the Chi2-algorithm for which the following condition is satisfied:

$$T_k \leq H_j < T_{k+1} \qquad k = 1, \ldots, K \tag{26}$$

$$\text{with } H_j = \sigma(\sum_{i=1}^{I} w_{ij} x_i) \tag{27}$$

and T_k and T_{k+1} thresholds determined by the Chi2-algorithm. This condition is equivalent to:

$$\sigma^{-1}(T_k) \leq \sum_{i=1}^{I} w_{ij} x_i < \sigma^{-1}(T_{k+1}) \qquad k = 1, \ldots, K \tag{28}$$

Thus conditions on the discretized activation values can be equivalently written as conditions on the weighted input variables.

In the last phase, the rules from the two previous phases are combined to form oblique classification rules in the original inputs.

3.3.5 FERNN

FERNN [44] is a decompositional algorithm for classification rule extraction from feedforward neural networks. The extraction process returns oblique rules, but under certain circumstances these rules can be simplified to M-of-N rules or DNF-rules.

FERNN requires the underlying network to have one hidden layer with sigmoidal activation functions. After training of this network, C4.5 is employed to identify the relevant hidden units. The samples used for construction of the decision tree are the hidden unit activations of the training patterns that were correctly classified by the network together with the corresponding class labels. Thus, from the correctly labelled examples a decision tree is created that predicts the class y^μ based on the vector of activation values $H_1^\mu, \ldots, H_J^\mu$ with J the number of hidden neurons and

$$H_j^\mu = \sigma(\sum_{i=1}^{I} w_{ij} x_i^\mu) \tag{29}$$

The test associated with each internal node n of this decision tree can be formulated as $H_j < threshold_n$. While it is relatively straightforward to convert the tree in a number of oblique rules, the conditions will include all of the input variables x_i, even if they are irrelevant for the classification performance. Setiono et al. [44] show that removal of inputs with small weights w_{ij} and an adapted threshold, can result in conditions that are equivalent to the original conditions but that are easier to comprehend as they contain only the relevant inputs. In a final step, it is proven that under very strict circumstances (e.g, binary inputs and weights lying in a limited interval), the conditions can be further simplified to M-of-N or DNF rules.

3.3.6 REFANN

REFANN (Rule Extraction from Function Approximating Neural Networks) [45] is a decompositional rule extraction algorithm for regression problems. As the name suggests, REFANN is limited to the extraction of rules from a specific type of neural network. The method was designed specifically for multilayer neural networks with one hidden layer. It assumes that the activation function used in the hidden layer is the hyperbolic tangent function $h(x) = tanh(x) = (e^x - e^{-x})/(e^x + e^{-x})$ and that no activation function was used for the output unit (Figure 17). The output of the network for an input sample $\mathbf{x}$ is therefore equal to:

$$\hat{y} = \sum_{j=1}^{H} v_j tanh(\xi_j) \tag{30}$$

$$\xi_j = \mathbf{W}_j \mathbf{x} \tag{31}$$

with H the number of hidden neurons, $\mathbf{W}_j$ the weight vector from the inputs to hidden unit j and v_j the connection weight from the hidden unit j towards the output unit.

The principal idea behind the algorithm is to approximate the hidden layer's activation functions by multiple linear functions as shown in Figure 17. Subsequently, these linear approximations are used to create the rules from. An overview of all steps necessary to generate the regression rules can be summarized as follows:

Step 1 Train and prune a neural network. Pruning of irrelevant input and hidden nodes is essential to create a network that generalizes well and to allow the extraction of a small set of comprehensible rules. The N2PFA (Neural Network Pruning for Function Approximation) algorithm is used for pruning the network. The main reason of selecting N2PFA over other pruning algorithms is that it removes complete neurons instead of only some connections. This will result in better comprehensibility because fewer inputs remain for inclusion in the rule-conditions.

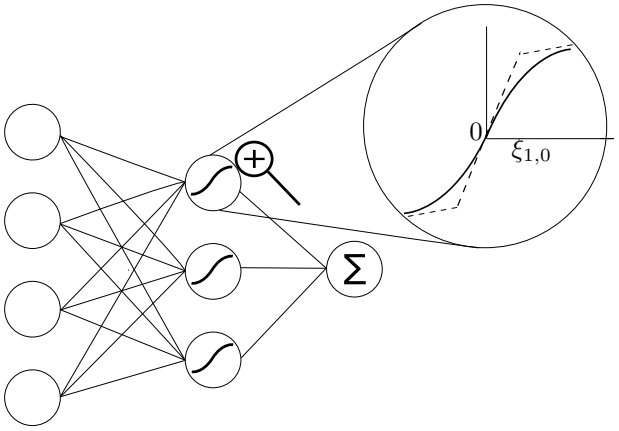


Figure 17: REFANN network and activation function approximation

Step 2 For each hidden unit i=1,...,H, locally **approximate the activation function** h(x) by a piecewise linear function $L_i(x)$. There are several possibilities to obtain this approximation. In [45], the piecewise approximation is constructed by minimizing the area bounded by $L_i(x)$ and h(x) on an interval centered around the origin. It is explained how this approximation can be calculated when the function $L_i(x)$ consists of respectively 3 (as in Figure 17) and 5 linear functions.

In [49], a different approach was followed to derive the piecewise linear approximation. It is ensured that the 3-piece linear approximation minimizes the sum of the squared errors for the training observations. This variant of the method is called NNRULES.

Whatever the exact method used to approximate the activation functions, the result of this step is a number of linear piecewise approximations for each of the hidden units' activation functions. A 3-piece approximation looks as follows:

$$L_i(x) = \begin{cases} -\alpha + \beta_1 x & if \quad x < -x_0 \\ \beta x & if \quad -x_0 \leq x \leq x_0 \\ \alpha + \beta_1 x & if \quad x > x_0 \end{cases} \tag{32}$$

Thus each approximation divides the input space to the neuron into 3 regions. Combining all the approximations of the hidden neurons therefore results in 3^H subregions.

Step 3 Generate the regression rules from the piecewise linear approximations. For each non-empty subregion, a rule is generated as follows:

- Define a linear equation that approximates the network's output in this subregion as the consequence of the rule

$$\hat{y} = \sum_{j=1}^{H} v_j L_j(\xi_j) \tag{33}$$

- Create the condition for this rule: (C_1 and C_2 and ... and C_H) where each C_i is either $\xi_i < -\xi_{i,0}$, $-\xi_{i,0} \leq \xi_i \leq -\xi_{i,0}$ or $\xi_i > \xi_{i,0}$ with $\xi_i = \mathbf{W}_i\mathbf{x}$ the input of hidden unit i and $\xi_{i,0}$ a constant obtained during the linear approximation process that represents the location of the kink in the piecewise approximation.

Step 4 (Optional) Apply C4.5 to **simplify the rule conditions**. As the conditions of the rules are based on scalar products involving the input weights, the resulting boundaries are oblique hyperplanes in the input space, e.g. $0.4\ \text{Input}_1 - 0.2\ \text{Input}_2 > 11$. To facilitate interpretation by analysts, it might be better to replace these oblique boundaries by ones that are parallel with the axes of the input space. C4.5 can be adopted for this task by assigning each training observation a class label corresponding to the subregion it belongs to. C4.5 is then able to learn parallel decision boundaries from these labelled observations that can be used to replace the original oblique conditions.

In [45], some empirical tests were performed with REFANN and the results show that the method is able to generate accurate results. There are however also some drawbacks related to this method. First, the number of extracted rules will grow exponentially with the number of hidden neurons. This means that the method is only applicable if the problem can be solved by a neural network that contains a limited number of hidden neurons.

The second drawback is associated with all dependent approaches. Whereas it is rather straightforward to alter REFANN's method for the approximation of the hidden unit activation function, it seems much more difficult to apply the algorithm when there are more drastic changes to the architecture of the neural network, e.g. networks with several hidden layers.

3.3.7 SVM+Prototypes

One of the few rule extraction methods designed specifically for support vector machines is the SVM+Prototypes method proposed in [36]. This decompositional algorithm is not only able to extract propositional (interval) classification rules from a trained SVM, but also rules from which the conditions are mathematical equations of ellipsoids. We will discuss the variant that results in propositional rules.

The SVM+Prototypes algorithm is an iterative process that proceeds as follows:

Step 1 Train a Support Vector Machine The SVM's decision boundary will divide the training data in two subsets $\mathcal{S}^+$ and $\mathcal{S}^-$, containing the instances for which the predicted class is respectively positive and negative.

Initialize the variable i to 1.

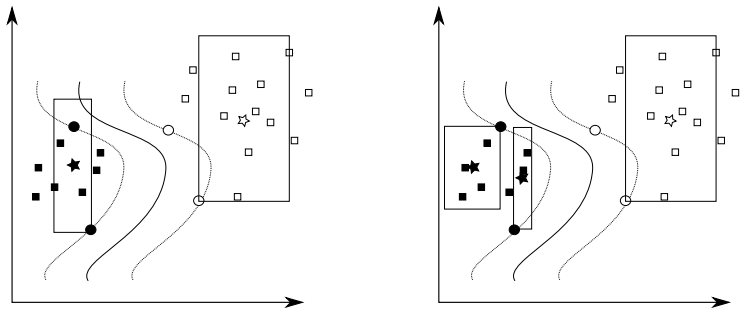
Step 2 For each subset, use some clustering algorithm to find i clusters (new subsets) and calculate the prototype or centroid of each cluster. For each of these new subsets find the support vector that lies farthest to the prototype. Use the prototype as center and the support vector as vertex to create a hypercube in the input space.

Step 3 Do a partition test on each of the hypercubes. This partition test is performed to minimize the level of overlapping between cubes for which the predicted class is different. One possible method is to test whether all of the corners of the hypercube are predicted to be of the same class. If this is the case then we say that the partition test is positive.

Step 4 Convert the hypercubes with a negative partition test into rules. If there are hypercubes with a positive partition test and i is smaller than a user-specified threshold I_{max} then increase i, take the subsets from which these cubes were created and go back to STEP 2, else go to STEP 5

Step 5 If i is equal to I_{max}, convert all of the current hypercubes into rules.

The example of Figure 18 shows the principal idea behind the algorithm. During the first iteration (i=1), the algorithm looks for the centroid of respectively the black and white instances. These prototypes are indicated by a star sign. It will then search the support vector in that partition that lies farthest away from the prototype and will create a cube from these two points (Step Two). In step three, the partition test will be positive for the leftmost cube as one of its vertices lies in the area for which the SVM predicts a different class. The other cube has a negative partition test and will therefore become a rule in step 4. For the first cube with a positive partition test, we will iterate the above procedure but with i=2. This will result in the creation of two new rules.



(a) First iteration (b) Second iteration

Figure 18: Example of SVM+Prototypes algorithm

The main drawback of this algorithm is that the extracted rules are neither exclusive nor exhaustive which results in conflicting or missing rules for the

classification of new data instances. Each of the extracted rules will also contain all possible input variables in its conditions, making the approach undesirable for larger input spaces as it will extract complex rules that lack interpretability. In [7] and [33], another issue with the scalability of this method is observed: a higher number of input patterns will result in more rules being extracted, which further reduces comprehensibility.

3.3.8 RN2

RN2 (Rule Extraction from Neural Networks, version 2) was proposed in [41] by Saito et al. as a decompositional technique for regression rule extraction. RN2 assumes that both numerical and nominal variables appear in the problem domain and creates regression rules where the conditional part is a logical formula over the nominal variables and the action part is expressed as a polynomial over the numeric variables [6]. An example of such a rule is:

if Battery="A" then Current=0.006+1.047 $(Conductance)^{0.964}$

It was shown that many phenomena in physics can be explained by this kind of equations.

RN2 assumes that the underlying neural network has a very specific architecture which severely limits the portability of this algorithm. The expected architecture is a network with one hidden layer and product units as the hidden neurons. Product units use $\prod_{i=1}^{I} x_i^{w_i}$ to combine the inputs instead of the more widespread summation units that calculate their input as $\sum_{i=1}^{I} w_i x_i$. The expected output of such product unit network is:

$$\hat{y} = v_0 + \sum_{j=1}^{H} v_j \prod_{i=1}^{I} x_i^{w_{ij}} = v_0 + \sum_{j=1}^{H} v_j \exp(\sum_{i=1}^{I} w_{ij} \ln x_i) \tag{34}$$

with I and H respectively the number of inputs and hidden neurons. We can extend the above model to allow nominal variables. Let I_{num} and I_{nom} be the respective number of numerical and nominal variables and let q_l be a nominal input variable with as possible values the elements of the set Q_l. For each q_l, a number of binary variables q_{lm} are introduced with q_{lm}=1 if q_l equals the m-th value in Q_l and else 0. Equation 34 can then be rewritten as:

$$\hat{y} = v_0 + \sum_{j=1}^{H} v_j \ exp(\sum_{l=1}^{I^{nom}} \sum_{m=1}^{M_l} w_{lmj} q_{lm}) \ exp(\sum_{i=1}^{I} w_{ij} \ln x_i) \tag{35}$$

by replacing

$$exp(\sum_{i=1}^{I} w_{ij} \ln x_i) = \prod_{i=1}^{I} x_i^{w_{ij}} \tag{36}$$

[6] A preceding algorithm, RF5, was presented in [40]. It assumes that no nominal variables appear in the problem domain.

and

$$exp(\sum_{l=1}^{I^{nom}} \sum_{m=1}^{M_l} w_{lmj} q_{lm}) = c_j \tag{37}$$

Equation 35 can be rewritten as:

$$\hat{y} = v_0 + \sum_{j=1}^{H} v_j c_j \prod_{i=1}^{I} x_i^{w_{ij}} \tag{38}$$

It is straightforward to extract regression rules from this equation. For example, we can create as many rules as there are training examples. For each training example μ=1,. . . ,N, the coefficients c_j^μ can be calculated from Equation 37 and a rule with polynomial action part is obtained as follows:

$$if \bigwedge_{l=1}^{I_{nom}} \bigvee_{q_{lm} \in Q_l^\mu} q_{lm} \ then \ \hat{y} = v_0 + \sum_{j=1}^{H} v_j c_j \prod_{i=1}^{I} x_i^{w_{ij}} \tag{39}$$

with $Q_l^\mu = \{q_{lm}^\mu : q_{lm}^\mu = 1\}$ a subset of Q_l that contains the nominal variables q_{lm} which are true for observation μ. It is however undesirable to extract a rule for each training observation as this will result in a large number of similar rules. Furthermore, the condition part of these rules will be very specific as they are based on only one training example.

To overcome this problem, the creators of RN2 suggested a clustering approach to find representative vectors that allow fewer and more general rules to be extracted. K-means clustering is performed on the vectors $\mathbf{c}^\mu = (c_1^\mu, \ldots, c_H^\mu)^T : \mu = 1, \ldots, N$ of the training examples to find this set of representative vectors: $\mathbf{r}^i = (r_1^i, \ldots, r_H^i)^T : i = 1, \ldots, K$ with K the number of representatives. Cross-validation is used to decide on the optimal number of representatives. For each representative a rule can easily be extracted as follows:

$$if \ i(\boldsymbol{q})=k \ \ then \ \hat{y} = v_0 + \sum_{j=1}^{H} v_j r_j^k \prod_{i=1}^{I} x_i^{w_{ij}}, k = 1, \ldots, K \tag{40}$$

with i($\mathbf{q}$) a function that returns the index of the representative vector $\mathbf{r}$ that lies closest to the vector $\mathbf{c}$ of the example. Formally, i($\mathbf{q}$) is defined as:

$$i(\mathbf{q}^\mu) = \arg\min_i \sum_{j=1}^{H} (c_j^\mu - r_j^i)^2 \tag{41}$$

A final step in the rule extraction process is to replace the conditional part of Equation 40 by a formula that is easier to understand. The approach that RN2 follows is similar to how REFANN eliminates its oblique decision boundaries. This task is performed by constructing a C4.5 decision tree from the training samples $(\mathbf{q}^\mu, i(\mathbf{q}^\mu))$ for $\mu = 1, \ldots, N$. From this decision tree, the conditions can then easily be extracted as a propositional rule with conditions on the nominal variables q_l.

In [41], the RN2-algorithm was applied on several artificial data sets and the algorithm was able to extract the rules that were embedded in these data sets. Additionally, experiments with real-life data showed that the method might also be useful for practical applications.

However, the method also faces some serious drawbacks. First, the polynomial conclusions of the rules only differ in their coefficients, while the exponents must be the same for each rule. A second negative point is the limited portability of the algorithm: RN2 requires the hidden neurons to be product units. This type of units are only rarely used and this requirement therefore seriously limits the practical applicability of this algorithm.

3.3.9 Barakat et al.

In [7], a dependent method for classification rule extraction from support vector machines was proposed. The algorithm is very similar to independent algorithms such as [6] that train from the original data with the output values replaced by the black box forecasts. The reason to classify this extraction method 'dependent' is the fact that only the instances corresponding to support vectors are used to extract rules from. In summary, in this approach we first train a support vector machine and select the instances that correspond to support vectors. From these instances, we replace the class labels by a class label that is predicted by the trained SVM. Afterwards, these instances are used as inputs to a 'white box' machine learning technique.

3.3.10 Fung et al.

In [21], Fung et al. present an algorithm to extract propositional classification rules from linear classifiers. The method is considered to be decompositional because it is only applicable when the underlying model provides a linear decision boundary. The resulting rules are parallel with the axes and non-overlapping, but only (asymptotically) exhaustive. Completeness can however be ensured by retrieving rules for only one of both classes and specification of a default class.

The algorithm is iterative and extracts the rules by solving a constrained optimization problem that is computationally inexpensive to solve. While the mathematical details are relatively complex and can be found in [21], the principal idea is rather straightforward to explain. Figure 19 shows execution of the algorithm when there are two inputs and when only rules for the black squares are being extracted.

First a transformation is performed such that all inputs of the black squares observations are in the interval [0,1]. Then the algorithm searches for an (hyper)cube that has one vertex on the separating hyperplane and lies completely in the region below the separating hyperplane. There are many cubes that satisfy these criteria, and therefore the authors added a criterion to find the 'optimal' cube. They developed two variants of the algorithm that differ only in the way this optimality is defined: volume maximization and point coverage

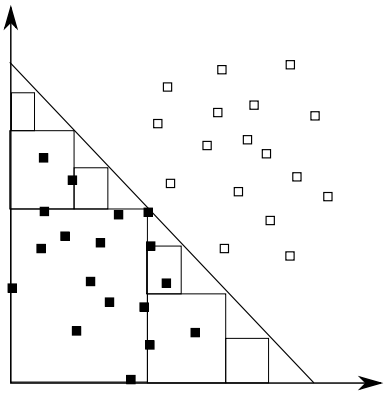


Figure 19: Example of algorithm of Fung et al.

maximization. The first variant retrieves the optimal cube as the cube that covers the largest possible volume, while the second variant defines the optimal cube as the cube that contains the largest possible number of training points. In the example of Figure 19, this 'optimal' cube is the large cube that has the origin as one of its vertices. This cube divides the region below the separating hyperplane in two new regions: the regions above and right of the cube. In general for an N-dimensional input space, one rule will create N new regions. In the next iteration, a new 'optimal' cube is recursively retrieved for each of the new regions that contain training observations. The algorithm stops after a user-determined maximum number of iterations.

The proposed method faces several drawbacks. Similarly to the SVM+Prototypes method discussed above, each rule condition involves all the input variables. This makes the method unsuitable for problems with a high-dimensional input space. A second limitation is the restriction to linear classifiers. This requirement considerably reduces the possible application domains.

4 Overview

A chronological overview of all discussed algorithms (and some additional techniques that were not discussed in the text) is given below in Table 4. For each algorithm, we provide the following information:

Type (I or D): **I**ndependent or **D**ependent

Scope (BC, C or R): **B**inary **C**lassification, **C**lassification or **R**egression

Summary: A very short description of the algorithm

Table 5 contains a chronological overview of papers that discuss rule extraction, but that do not introduce a new algorithm or technique.

Algorithm(Year)	Ref.	Type	Scope	Summary
CART(1984)	[9]	I	C+R	-decision tree induction
CN2 (1989)	[11]	I	C	-rule induction
KT (1991)	[20]	D	C	-find subsets of weights that ensure activation of the output unit
C4.5(1993)	[37]	I	C	-decision tree induction
SUBSET(1993)	[56]	D	C	-find subsets of weights that ensure activation of the output unit
N-of-M(1993)	[56]	D	C	-create M-of-N rules, based on clustering of similar weights
VIA(1993)	[52]	D	C(R)	-generate rules and test whether they are consistent with the neural network by Interval Analysis
REAL(1994)	[14]	D	C	-create a very specific rule covering one example, iteratively remove conditions and test whether rule is still correct (e.g. with VIA-method)
TREPAN(1996)	[15]	I	C	-decision tree induction, M-of-N splits
NeuroLinear(1997)	[48]	D	C	-oblique rules, applicable to neural networks with one hidden layer, discretization of hidden unit activation values is followed by a two-phase extraction process
RF5(1997)	[40]	D	R	-predecessor of RN2, accepts only numeric inputs
BIO-RE(1999)	[51]	I	BC	-creates complete truth table, only applicable to toy problems
PARTIAL-RE(1999)	[51]	D	C	-similar to SUBSET, uses weight ordering
FULL-RE(1999)	[51]	D	C	-similar to SUBSET, uses linear programming
ANN-DT(1999)	[42]	I	C+R	-decision tree induction, similar to TREPAN
FERNN(2000)	[44]	D	C	-oblique rules (also M-of-N and DNF), applies C4.5 on hidden unit activation values and their corresponding target
DecText(2000)	[8]	I	C	-decision tree induction
RN2(2002)	[41]	D	R	-polynomial conclusions, RN2 assumes product units in hidden units and clusters the activation values of the hidden units
REFANN(2002)	[45]	D	R	-approximation of NN activation functions by piecewise linear functions
SVM+Prototypes(2002)	[36]	D	C	-clustering
STARE(2003)	[61]	I	C	-breadth-first search with sampling, prefers categorical variables over continuous variables
G-REX(2003)	[25]	I	C+R	-genetic programming: different types of rules
REX(2003)	[32]	I	C	-genetic algorithm: fuzzy rules
GEX(2004)	[31]	I	C	-genetic algorithm: propositional rules
NNRULES(2004)	[49]	D	R	-approximation of NN activation functions by piecewise linear functions
Rabuñal(2004)	[38]	I	C	-genetic programming
BUR(2004)	[10]	I	C	-based on gradient boosting machines
Barakat(2005)	[7]	D	C	-train decision tree on support vectors and their class labels
Fung(2005)	[21]	D	BC	-only applicable to linear classifiers
ITER(2006)	[23]	I	C+R	-iterative growing of hypercubes

Table 4: Chronological Overview of Rule Extraction Algorithms

Lead Author (reference)	year	Algorithms Covered	Comments
Andrews [3]	1995	RULENET, SUBSET, M-of-N, REAL, VIA, RULEX, BRAINNE	-introduces the ADT-Taxonomy
Neumann [35]	1998	VIA, BIO-RE, RULENEG, NEUROLINEAE,PARTIAL-RE, SUBSET, M-of-N, FULL-RE, RULEX, RULENET, REAL, DEDEC	-review of algorithms according to ADT-Taxonomy
Tickle [55]	1998	COMBO, RF5, RX, IA, TREPAN, TOPGEN	-refines ADT-Taxonomy and provides a survey of algorithms
Craven [16]	1999	TREPAN	-provides recommendations and directions for researchers on Rule Extraction
Darbari [17]	2001	SUBSET, M-of-N, TREPAN, VIA	-discusses hybrid learning, contains some experimental results
Duch [18]	2004	VIA, SUBSET, M-of-N, RULENET, REAL, RULENEG, BRAINNE, DEDEC, TREPAN, RULEX	-discusses a.o. rule types and rule optimization, contains short overview of algorithms
Kordos [30]	2005	VIA, TREPAN, RULENEG, BIO-RE, PARTIAL-RE, RX, SUBSET, M-of-N, RULEX, NEURORULE, FERNN, NEFCLASS, GEX	-PhD thesis with a chapter that describes various neural network extraction algorithms
Jacobsson [24]	2005	-	-introduces a taxonomy for rule extraction from recurrent neural networks and provides a detailed overview of algorithms
Martens [33]	2005	SVM+Prototype, Fung et al., C4.5, TREPAN, G-REX	-small benchmarking study, specifically targets SVM rule extraction

Table 5: Overview papers of Rule Extraction

5 Conclusion

Although the success stories of opaque predictive models in research have triggered interest from business practitioners, the widespread adoption of these techniques in business applications seems to lag behind. The difficulties associated with a correct architecture and parameter selection might serve as a partial explanation for this slow adoption rate, but the main reason is probably the lack of interpretability of these models.

Whereas a linear model is limited in its predictive accuracy, it has the advantage that analysts can immediately see what the impact of a variable is on the target variable. This is in sharp contrast with opaque models, such as neural networks and support vector machines, for which the relation between input variables and the target concept is difficult to grasp. While these models can offer better performance, the increase is often not enough to compensate for their opaqueness. The ability to extract human-comprehensible descriptions from these models might therefore facilitate their adoption for applications where comprehensibility is a key requirement.

In this report, we provided an in-depth overview of several algorithms that have the ability to extract such understandable descriptions from a trained black box model. Our main conclusions after reviewing all these algorithms can be summarized in several points.

• A first conclusion is that there are only very few methods that are developed specifically to extract rules from models other than neural networks. For example, only two dependent algorithms [7, 36] were found that are designed specifically for support vector machines and only a single algorithm seems to take the characteristics of ensemble methods into account [58]. This is in deep contrast with neural networks, for which a plethora of dependent algorithms is available. It is therefore useful to see to what extent these dependent methods must be altered to make them suitable for rule extraction from support vector machines or ensembles. Due to the close intertwining of these dependent algorithms with the underlying neural networks, we believe however that in most cases such conversions will be hard or even impossible. The consequence is that just a few algorithms remain applicable whenever the underlying model is not a neural network. Only independent algorithms have the flexibility to deal with a wide variety of underlying models.

• A second observation is the lack of executable code for most of the algorithms. In [16], it was already expressed that availability of software is of crucial importance to achieve a wide impact of rule extraction. However, only few algorithms are publicly available. This makes it difficult to gain an objective view of the algorithms' performance or to benchmark multiple algorithms on a data set. Furthermore, we are convinced that it is not only useful to make the completed programs available, but also to provide code for the subroutines used within these programs as they can often be shared. For example, the creation of artificial observations in a constrained part of the input space is a routine that is used by several methods, e.g. TREPAN, ANN-DT and ITER. Other routines that can benefit from sharing and that can facilitate development of

new techniques are procedures to query the underlying model or routines to optimize the returned rule set.

• The last observation, but probably the most important, is that several authors seem to have forgotten that the goal of rule extraction is dual, the extracted description should be both accurate and comprehensible. Several papers put the emphasis on the accuracy of the extracted rule set, but never check whether the extracted description is really more comprehensible than the original model. In several papers the number of extracted rules is so large that we are afraid that the lack of interpretability of the black box is replaced by a set of rules that is even less comprehensible. Only very limited research has been performed on this aspect of rule extraction.

In this research report, we believe that our contributions can be summarized as threefold:

• First, we developed a method that can be used to measure the consistency of an extraction algorithm. The method is quite general and therefore it is applicable to a wide range of representations. The introduction of this measure will hopefully help in drawing attention to this often overlooked aspect.

• Second, we created a taxonomy that allows for a clear categorization of rule extraction algorithms. Whereas, the widespread ADT-Taxonomy was specifically developed for rule extraction from neural networks, we believe that our taxonomy is suitable for a much broader range of algorithms.

• Finally, we provided an up-to-date review of the most prominent rule extraction algorithms. This review helped us to identify the main shortcomings and provided us with ideas for the extraction algorithm that we present in [23].

References

[1] Equal Credit Opportunity Act, United States Code, title 15, chapter 41, subchapter IV, 1974.

[2] R. Andrews. *An Automated Rule Refinement System.* PhD thesis, Queensland University of Technology, 2003.

[3] R. Andrews, J. Diederich and A. Tickle. Survey and critique of techniques for extracting rules from trained artificial neural networks. *Knowledge Based Systems*, 8(6):373–389, 1995.

[4] B. Baesens. *Developing Intelligent Systems for Credit Scoring Using Machine Learning Techniques.* PhD thesis, Katholieke Universiteit Leuven, Faculteit Economische en Toegepaste Economische Wetenschappen, 2003.

[5] B. Baesens, T. Van Gestel, S. Viaene, M. Stepanova, J. Suykens and J. Vanthienen. Benchmarking state of the art classification algorithms for credit scoring. *Journal of the Operational Research Society*, 54(6):627–635, 2003.

[6] N. Barakat and J. Diederich. Learning-based rule-extraction from support vector machines. *In The 14th International Conference on Computer Theory and applications ICCTA'2004*, 2004.

[7] N. Barakat and J. Diederich. Eclectic rule-extraction from support vector machines. *International Journal of Computational Intelligence*, 2(1):59–62, 2005.

[8] O. Boz. *Converting A Trained Neural Network To A Decision Tree DecText - Decision Tree Extractor*. PhD thesis, Lehigh University, Department of Computer Science and Engineering, 2000.

[9] L. Breiman, J.H. Friedman, R.A. Olsen and C.J. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, 1984.

[10] F. Chen. Learning accurate and understandable rules from SVM classifiers. Master's thesis, Simon Fraser University, 2004.

[11] P. Clark and T. Niblett. The CN2 induction algorithm. *Machine Learning*, 3:261–283, 1989.

[12] W. Cohen. Fast effective rule induction. In Armand Prieditis and Stuart Russell, editors, *Proc. of the 12th International Conference on Machine Learning*, pages 115–123, Tahoe City, CA, 1995. Morgan Kaufmann.

[13] M.W. Craven. *Extracting Comprehensible Models from Trained Neural Networks*. PhD thesis, Department of Computer Sciences, University of Wisconsin-Madison, 1996.

[14] M.W. Craven and J.W. Shavlik. Using sampling and queries to extract rules from trained neural networks. *In International Conference on Machine Learning*, pages 37–45, 1994.

[15] M.W. Craven and J.W. Shavlik. Extracting tree-structured representations of trained networks. In David S. Touretzky, Michael C. Mozer and Michael E. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8, pages 24–30. The MIT Press, 1996.

[16] M.W. Craven and J.W. Shavlik. Rule extraction: Where do we go from here? Working paper, University of Wisconsin, Department of Computer Sciences, 1999.

[17] A. Darbari. Rule extraction from trained ann: a survey. Technical report, TU Dresden, Institute of Artificial Intelligence, Department of Computer Science, 2001.

[18] W. Duch, R. Setiono and J.M. Zurada. Computational intelligence methods for rule-based data understanding. *Proceedings of the IEEE*, 92(5):771–805, 2004.

[19] J.H. Friedman. Greedy function approximation: A gradient boosting machine. Technical report, Stanford University, Department of Statistics, 1999.

[20] L. Fu. Rule learning by searching on adapted nets. *In Ninth National Conference on Artificial Intelligence*, pages 590–595, 1991.

[21] G. Fung, S. Sandilya and R.B. Rao. Rule extraction from linear support vector machines. *In 11th ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 32–40, 2005.

[22] D. Hand, H. Mannila and P. Smyth. *Principles of Data Mining*. MIT Press, 2001.

[23] J. Huysmans, B. Baesens and J. Vanthienen. ITER: an algorithm for predictive regression rule extraction. *In 8th International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2006)*, pages –. Springer Verlag, lncs, 2006.

[24] H. Jacobsson. Rule extraction from recurrent neural networks: A taxonomy and review. *Neural Computation*, 17:1223–1263, 2005.

[25] U. Johansson, R. König and L. Niklasson. Rule extraction from trained neural networks using genetic programming. *In Joint 13th International Conference on Artificial Neural Networks and 10th International Conference on Neural Information Processing, ICANN/ICONIP 2003*, pages 13–16, 2003.

[26] U. Johansson, R. König and L. Niklasson. The truth is in there - rule extraction from opaque models using genetic programming. *In FLAIRS Conference*, 2004.

[27] U. Johansson, R. König and L. Niklasson. Automatically balancing accuracy and comprehensibility in predictive modeling. *In Proceedings of the 8th International Conference on Information Fusion*, 2005.

[28] A. Karalic. Linear regression in regression tree leaves. *In ISSEK '92 (International School for Synthesis of Expert Knowledge)*, pages 151–163, 1992.

[29] R. Kohavi and J.R. Quinlan. Decision-tree discovery. In W. Klosgen and J. Zytkow, editors, *Handbook of Data Mining and Knowledge Discovery*, pages 267–276. Oxford University Press, 2002.

[30] M. Kordos. *Search-based Algorithms for Multilayer Perceptrons*. PhD thesis, The Silesian University of Technology, 2005.

[31] U. Markowska-Kaczmar and M. Chumieja. Discovering the mysteries of neural networks. *International Journal of Hybrid Intelligent Systems*, 1(3-4):153–163, 2004.

[32] U. Markowska-Kaczmar and W. Trelak. Extraction of fuzzy rules from trained neural network using evolutionary algorithm. *In European Symposium on Artificial Neural Networks (ESANN)*, pages 149–154, 2003.

[33] D. Martens, B. Baesens, T. Van Gestel and J. Vanthienen. Adding comprehensibility to support vector machines using rule extraction techniques. *In Credit Scoring and Credit Control IX*, 2005.

[34] R. Michalski. On the quasi-minimal solution of the general covering problem. *In Proceedings of the V International Symposium on Information Processing (FCIP 69)*, pages 125–128, 1969.

[35] J. Neumann. Classification and evaluation of algorithms for rule extraction from artificial neural networks. PhD summer project, University of Edingburgh, 1998.

[36] H. Núñez, C. Angulo and A. Català. Rule extraction from support vector machines. *In European Symposium on Artificial Neural Networks (ESANN)*, pages 107–112, 2002.

[37] J.R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[38] J.R. Rabuñal, J. Dorado, A. Pazos, J. Pereira and D. Rivero. A new approach to the extraction of ANN rules and to their generalization capacity through GP. *Neural Computation*, 16(47):1483–1523, 2004.

[39] K. Saito and R. Nakano. Medical diagnostic expert system based on pdp model. *In Proceedings of IEEE International Conference on Neural Networks*, volume 1, pages 255–262, 1988.

[40] K. Saito and R. Nakano. Law discovery using neural networks. *In Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 1078–1083, 1997.

[41] K. Saito and R. Nakano. Extracting regression rules from neural networks. *Neural Networks*, 15(10):1279–1288, 2002.

[42] G.P.J. Schmitz, C. Aldrich and F.S. Gouws. ANN-DT: An algorithm for extraction of decision trees from artificial neural networks. *IEEE Transactions on Neural Networks*, 10(6):1392–1401, 1999.

[43] R. Setiono and B. Baesens. Risk management using recursive neural network rule extraction. *Submitted to Management Science*, 2006.

[44] R. Setiono and W.K. Leow. FERNN: An algorithm for fast extraction of rules from neural networks. *Applied Intelligence*, 12(1-2):15–25, 2000.

[45] R. Setiono, W.K. Leow and J.M. Zurada. Extraction of rules from artificial neural networks for nonlinear regression. *IEEE Transactions on Neural Networks*, 13(3):564–577, 2002.

[46] R. Setiono and H. Liu. Chi2: Feature selection and discretization of numeric attributes. *In Proceedings of the 7th IEEE International Conference on Tools with Artificial Intelligence*, pages 388–391, 1995.

[47] R. Setiono and H. Liu. X2R: A fast rule generator. *In Proceedings of the 7th IEEE International Conference on Systems, Management and Cybernetics*, 1995.

[48] R. Setiono and H. Liu. Neurolinear: From neural networks to oblique decision rules. *Neural Computing*, 17(1):1–24, 1997.

[49] R. Setiono and J.Y.L. Thong. An approach to generate rules from neural networks for regression problems. *European Journal of Operational Research*, 155(1):239–250, 2004.

[50] D.W. Silverman. *Density Estimation for Statistics and Data Analysis*. Chapman and Hall, 1986.

[51] I. Taha and J. Ghosh. Symbolic interpretation of artificial neural networks. *IEEE Transactions on Knowledge and Data Engineering*, 11(3):448–463, 1999.

[52] S. Thrun. Extracting provably correct rules from artificial neural networks. Technical report iai-tr-93-5, Universität Bonn, Institut fur Informatik III, 1993.

[53] S. Thrun. Extracting rules from artificial neural networks with distributed representations. In G. Tesauro, D. Touretzky and T. Leen, editors, *Advances in Neural Information Processing Systems (NIPS) 7*, Cambridge, MA, 1995. MIT Press.

[54] S. Thrun et al. The MONK's problems: A performance comparison of different learning algorithms. Technical Report CS-91-197, Pittsburgh, PA, 1991.

[55] A.B. Tickle, R. Andrews, M. Golea and J. Diederich. The truth will come to light: directions and challenges in extracting the knowledge embedded within trained artificial neural networks. *IEEE Transactions on Neural Networks*, 9(6):1057–1068, 1998.

[56] G. Towell and J. Shavlik. The extraction of refined rules from knowledge-based neural networks. *Machine Learning*, 13(1):71–101, 1993.

[57] S. Viaene, R. Derrig, B. Baesens and G. Dedene. A comparison of state-of-the-art classification techniques for expert automobile insurance fraud detection. *Journal of Risk And Insurance (Special Issue on Fraud Detection)*, 69(3):433–443, 2002.

[58] R. Wall, P. Cunningham and P. Walsh. Explaining predictions from a neural network ensemble one at a time. *In PKDD 2002*, pages 449–460, 2002.

[59] Z.-H. Zhou. Rule extraction: Using neural networks or for neural networks? *Journal of Computer Science and Technology*, 19(2):249–253, 2004.

[60] Z.-H. Zhou, S.-F. Chen and Z.-Q. Chen. A statistics based approach for extracting priority rules from trained neural networks. *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks*, 3, 2000.

[61] Z.-H. Zhou, Y. Jiang and S.-F. Chen. Extracting symbolic rules from trained neural network ensembles. *AI Communications*, 16(1):3–15, 2003.

A Gradient Descent

Gradient Descent, also known as Steepest Descent, is an optimization algorithm for finding the local minimum of a function. Starting from an initial position $\mathbf{X}_0$, the algorithm will move from $\mathbf{X}_i$ to $\mathbf{X}_{i+1}$ by going in the opposite direction of the gradient evaluated at the current point. Mathematically this can be formulated as:

$$X_{i+1} = X_i - \epsilon \nabla f(X_i) \tag{42}$$

with ϵ a small positive parameter. By replacing the above minus by a plus sign, i.e. taking steps proportional to the gradient, one approaches a local maximum of that function; the procedure is then known as gradient ascent.

If ϵ is chosen sufficiently small, and if there exists a local minimum, the above update rule will converge to this minimum after a finite number of iterations. Often, a suitable value for ϵ will be selected in advance and remain constant over time. Alternatively, we can select for each iteration the value of ϵ_i that minimizes:

$$\epsilon_i = \arg\min_{\epsilon} f(X_{i+1}) = f(X_i - \epsilon \nabla f(X_i)) \tag{43}$$

This optimization, also known as line search, ensures that the best value for ϵ is selected, but solving the optimization might be time-consuming and for some problems it is therefore actually slower than working with a fixed value.

There are two main drawbacks associated with Gradient Descent. For functions with plateaus Gradient Descent will require a large number of iterations as the gradient will be small. Second, the solution found by the Gradient Descent algorithm is only a local optimum and can vary depending on the starting location of the search.